UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Mention: Informatique

présentée et soutenue par

Bassem Ouni

High-level energy characterization, modeling and estimation for OS-based platforms

Thèse dirigée par *Cécile* Belleudy

soutenue le 11 juillet 2013

Jury :

M. Smail Niar	Professeur, Université de Valenciennes	Rapporteur
M. Yvon Trinquet	Professeur, Université de Nantes	Rapporteur
M. Eric Senn	Maître de Conférences, Université de Bretagne Sud	Examinateur
M. François Verdier	Professeur, Université de Nice-Sophia Antipolis	Examinateur
M. Sébastien Bilavarn	Maître de Conférences, Université de Nice-Sophia Antipolis	Examinateur
Mme. Cécile Belleudy	Maître de Conférences, Université de Nice-Sophia Antipolis	Directeur de Thèse

► TO MY PARENTS ◄

Acknowledgements

The road to a PhD is not always smooth, is sometimes hilly and has bends but its a pleasant journey with the help of the guides who know about the road, the other fellow travelers and the supporters. Here, I would like to thank all the people who contributed to this thesis.

I feel privileged to have worked with my advisor, Dr. Cécile BELLEUDY, who guided me with great enthusiasm and patience. She gave me freedom to explore and discover new areas in the domain of energy consumption of embedded systems and I have learnt a lot while working under her supervision. I am also grateful to Dr. Sébastien BILAVARN for providing the constant guidance and the helpful feedback on my work. Working in the MCSOC team at LEAT, Sophia Antipolis had been an unforgettable and very pleasant experience of my life, and I am going to miss the working environment here. I would also like to thank my colleagues for numerous useful discussions.

I feel honored to have respected researchers who served on my dissertation committee. I would like to thank my reviewing committee members: Dr. Smail NIAR and Dr. Yvon TRINQUET for their time, interest, and helpful comments. I would also like to thank my examiners, who provided encouraging and constructive feedback. It is no easy task, reviewing a thesis, and I am grateful for their thoughtful and detailed comments. To the many anonymous reviewers at the various conferences and journals, thank you for helping to shape and guide the direction of the work with your informative and instructive comments.

This thesis was funded by National Research Agency (ANR) of France in the frame of the OPEN-PEOPLE project. As a member of OPEN-PEOPLE, I have been surrounded by wonderful colleagues who have provided me a rich and fertile environment to study and explore new ideas. I would like to thank the project leader, Dr. Eric SENN, and Dr. Rabie BEN ATITALLAH who has been extremely supportive in allowing me to participate in LAMIH laboratory activities while pursuing my PhD studies.

A special thanks to all my friends, who have accompanied me in this wonderful journey of professional and personal growth that started in HORBIT, SIDI KHELIF (SIDI BOUZID) and ended in NICE. Thanks for putting up with me, being a support and sharing some unforgettable moments.

Lastly, I would like to thank my family for all their love and encouragement. I wish to thank my parents. They bore me, raised me, supported me, taught me and loved me. And most of all for my supportive, encouraging and patient fiancee whose faithful support during the final stages of this Ph.D. is so appreciated. I love you all dearly. Thank you.

Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 Contributions . . . 4
1.3 Outline . . . 6

2 Background on embedded systems energy consumption characterization, modeling and analysis **7**

2.1 Power and energy dissipation in embedded systems . . . 7
2.2 Power and energy consumption characterization and estimation in embedded systems . . . 9
2.2.1 Power and energy in electrical circuits . . . 9
2.2.2 Overview of power and energy consumption characterization and estimation . . . 10
2.3 Power and energy consumption estimation of embedded systems at different abstraction levels . . . 11
2.3.1 Estimation and modeling of power/energy consumption at microprocessor abstraction levels . . . 13
2.3.2 Power/energy consumption of hardware components and peripheral devices . . . 20
2.3.3 Characterization of embedded OS power/energy consumption 22
2.4 Conclusion . . . 30

3 Characterization and analysis of embedded OS services energy consumption **31**

3.1 Introduction . . . 32
3.2 Overview of embedded OS . . . 32
3.2.1 OS middleware in embedded systems . . . 32
3.2.2 Embedded OS services and functionalities . . . 33
3.3 Experimental setup . . . 38
3.3.1 OMAP3530 Applications Processor . . . 39
3.3.2 OMAP3530 EVM board . . . 39
3.3.3 Measurement framework . . . 39
3.4 Energy characterization and estimation flow . . . 41
3.5 OS power and energy modeling . . . 44
3.5.1 Scheduling routines . . . 45
3.5.2 Context switch . . . 46

3.5.3 Inter-process communication . . . 53
3.6 Conclusion . . . 56

4 High level modeling of embedded system components 59
4.1 Exploitation of high level AADL models . . . 60
4.2 Embedded OS functional/non-functional properties and requirements 60
4.3 Architecture modeling languages . . . 61
4.4 Overview of AADL language . . . 63
4.4.1 AADL components . . . 63
4.4.2 Subcomponents . . . 64
4.4.3 Components implementations . . . 65
4.4.4 Components interaction . . . 65
4.4.5 AADL properties, annexes, packages and modes . . . 66
4.4.6 AADL tools . . . 66
4.5 AADL modeling case study . . . 67
4.5.1 H.264 application . . . 67
4.5.2 AADL modeling of system components . . . 68
4.6 Conclusion . . . 77

5 Embedded OS service's models integration in the system level design flow 79
5.1 Models integration in multiprocessor scheduling simulation tool . . . 80
5.1.1 STORM tool . . . 80
5.1.2 The proposed approach . . . 81
5.2 Low power scheduling policies . . . 83
5.2.1 The AsDPM scheduling policy: . . . 84
5.2.2 The DSF scheduling policy: . . . 87
5.3 Embedded OS services energy overhead: . . . 88
5.3.1 Fixed frequency case: . . . 89
5.3.2 Dynamic frequency case: . . . 89
5.4 Experimental results: . . . 91
5.5 Conclusion . . . 94

6 System design space exploration and verification of constraints 97
6.1 AADL exploration of hardware software solutions . . . 97
6.2 Design space exploration methodology . . . 98
6.3 System constraints definition and verification flow . . . 99
6.3.1 RDAL Language and RDALTE tool . . . 100
6.3.2 The Object Constraint Language (OCL) . . . 100
6.3.3 The Quantitative Analysis Modeling Language (QAML) . . . 101
6.3.4 The proposed approach . . . 101

6.4 System requirements analysis, definition and verification 101
6.4.1 Quantitative analysis specifications using the QAML language 101
6.4.2 Requirements definition and verification using RDALTE tool 104
6.5 Example 109
6.6 Conclusion 111

7 Conclusion **113**
7.1 Conclusion 113
7.2 Perspectives 114
7.2.1 OS services energy characterization approach extension 115
7.2.2 OS services energy optimization 115
7.2.3 System level thermal modeling 115

Bibliography **117**

List of Figures

1.1 The use of operating systems in embedded systems development [111] 3
1.2 The reasons for non-use of operating systems in embedded projects [72] 3

2.1 Energy/power consumption estimation and characterization flow . . 12
2.2 Different abstraction levels 13
2.3 Ahuja et al.'s RTL power estimation flow [24] 18
2.4 Overview of Zhao et al. OS routines energy estimation approach [117] 25
2.5 Implementation of API and cache management functions [61] 27
2.6 Power synthesis, optimization and analysis methodology [56] 28

3.1 The different layers of an embedded system 33
3.2 Address conversion by memory management unit 35
3.3 Transitions between task states 36
3.4 The exploitation of hardware components by the OS services 38
3.5 OMAP3530 Applications Processor [7] 40
3.6 OMAP35x EVM Board top overview 41
3.7 OMAP35x EVM Board bottom overview 42
3.8 The measurement framework 43
3.9 The methodology of OS energy characterization 43
3.10 Estimation of the operating system energy consumption 44
3.11 Scheduling routines power consumption versus the number of processes for different scheduling policies 47
3.12 Scheduling routines energy variation as a function of CPU frequency (*SCHED_OTHER* policy and 10 processes) 48
3.13 Step 1 49
3.14 Step n 49
3.15 Context switch energy consumption versus the number of context switching for different scheduling policies 51
3.16 Context switch power variation as a function of CPU frequency . . . 52
3.17 Context switch energy variation as a function of dynamic CPU frequency scaling 53
3.18 Context switch power variation as a function of dynamic CPU frequency scaling 54
3.19 Context switch time variation as a function of dynamic CPU frequency scaling 54
3.20 IPC power variation as a function of CPU frequency 55
3.21 IPC energy variation as a function of message size 56

4.1 From AADL modeling to scheduling policies simulation and system requirements verification . . . 60
4.2 Graphical representations of software AADL components . . . 64
4.3 Graphical representations of hardware and generic AADL components 65
4.4 Block diagram of H.264 decoding scheme slices version . . . 69
4.5 Thread *NAL_DISPATCH* AADL model . . . 70
4.6 OS services AADL model . . . 71
4.7 Thread group of OS services . . . 72
4.8 AADL modeling of the communication between application and OS services . . . 73
4.9 angle=-90 . . . 74
4.10 AADL modeling of the OMAP3530 processor . . . 75
4.11 AADL implementation of system modes and events . . . 76
4.12 AADL model of software application on hardware platform . . . 77

5.1 STORM simulator input and output file system [9] . . . 81
5.2 Example of STORM input XML file . . . 82
5.3 Os power and energy models integration in the system level design flow 83
5.4 Task period extraction . . . 84
5.5 DPM technique energy saving . . . 85
5.6 Slack reclamation using the DSF technique . . . 89
5.7 Os calls at fixed frequency . . . 90
5.8 Os calls when changing the frequency . . . 91
5.9 Schedule of application tasks using DSF technique . . . 93
5.10 OS services and standalone application tasks energy consumption (DSF technique) . . . 94

6.1 Possible solution of AADL software tasks binding on hardware platform . . . 98
6.2 System design exploration methodology . . . 99
6.3 System constraints definition and verification flow . . . 102
6.4 Energy consumption estimation law of *slice*2*_processing* thread running at 720 Mhz . . . 103
6.5 Energy consumption composition law . . . 104
6.6 Evaluation of energy consumption composition law . . . 104
6.7 System requirements RDAL diagram . . . 105
6.8 Energy consumption requirement verification using OCL . . . 106
6.9 OCL expression of the schedulability test . . . 108
6.10 Requirements satisfaction rates (500 Mhz) . . . 110
6.11 Satisfaction rates of third level of exploration requirements . . . 111

List of Tables

3.1 Inter-process communication power models according to processor frequency 56
3.2 Inter-process communication energy models according to message size 56

4.1 AADL subcomponents 65
4.2 H.264 video decoder application tasks features 69

5.1 Power-efficient states of OMAP3530 processor @ 125-MHz 87
5.2 Voltage-frequency levels of OMAP330 processor 90
5.3 OS services energy consumption rates when using the DSF technique 94
5.4 OS services energy consumption rates when using the AsDPM technique 94

6.1 Characteristics of possible solutions 110

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Contributions 4
1.3 Outline 6

In this thesis, we focused on power and energy characterization, modeling, estimation and optimization of embedded systems running applications with operating system (OS) support. An energy consumption characterization flow is introduced and power/energy models of embedded OS services are extracted. Several hardware and software parameters are varied to estimate the embedded OS energy consumption. The obtained models are first integrated in a simulation tool for multiprocessor scheduling to calculate the OS energy consumption . Then, we compare the overhead of the embedded OS using low power scheduling policies such as DVFS (Dynamic Voltage Frequency Scaling) and DPM (Dynamic power management) techniques. Finally, system design exploration flow is introduced. We define and verify system requirements, using a set of tools: RDALTE and QAML, when allocating applicative tasks to processors. In this chapter, we begin by introducing the context of this thesis, then the main contributions, and we wrap up by presenting the outline of the dissertation.

1.1 Context

Nowadays technological developments have changed our lives. This has led to a rise of the automation in everyday life activities which is expected to increase further in the future. This automation is provided by computing systems that associate the electronic design and hardware components with software applications which are known as embedded systems. An embedded system is a microprocessor-based system that is incorporated into a device to monitor and control the functions of the components of this device [60]. The realm of embedded systems has expanded so that they are used in a wide variety of domains ranging from home appliances, wireless communications systems, medical imaging systems, automotive/transportation

devices to complex applications in avionics and defense.
Generally, embedded systems are composed of two main layers: a hardware layer, which is the physical support containing the hardware components, and the software layer represented by the operating system joined with the application. The physical support is the architecture containing all major physical components that execute the application tasks through the operating system services. The application and the operating system, a set of software tasks and services managing these tasks and the system resources, constitute the software part of an embedded system. Both hardware and software elements are closely linked and they are not easily discernible.
Embedded systems are one of the most efficient tools that can be used to resolve challenges faced when designing a new system. The architecture of an embedded device allows the definition of the design's infrastructure, constraints and options. In fact, embedded system interacts tightly with its environment: it respects physical phenomena constraints, real-time constraints, constraints of dynamism, energy consumption constraints, density constraints, robustness constraints, cost constraints, etc. Software, hardware and mixed architectures are efficiently designed to satisfy these constraints.
Embedded systems become complex as they contain various hardware devices and software applications which interact with the users to handle the system. The complexity of the hardware and software layers necessitates the use of a specific support allowing application to exploit efficiently the hardware platform. This support is the operating system (OS).
According to a recent poll data [111], as showed in figure 1.1, a little less than 73% of embedded systems engineers, programmers and managers around the world use the embedded OS for their projects. These statistics confirm the importance and wide spreading of embedded OS. This is thanks to the wide variety of its services and capabilities allowing an efficient exploitation of hardware resources.

The ever-increasing complexity of embedded systems that are developing their computation performances, ranging from multimedia and telecommunication to medical systems, poses a great challenge for embedded systems developers and experts: power and energy consumption. In fact, power densities in microprocessors have almost doubled every three years [47], [104]. Also, leakage power is increasing exponentially with process technology and it is expected to dominate the total power consumption. This growth in power consumption poses two main problems: the increase of energy consumption and device's temperature dissipation. For this reason, characterizing and reducing energy consumption of embedded systems is an important design challenge for embedded system designers. As the OS is a basic component of an embedded system, it has become imperative to take its energy overhead into account when designing embedded systems. In figure 1.2, we show

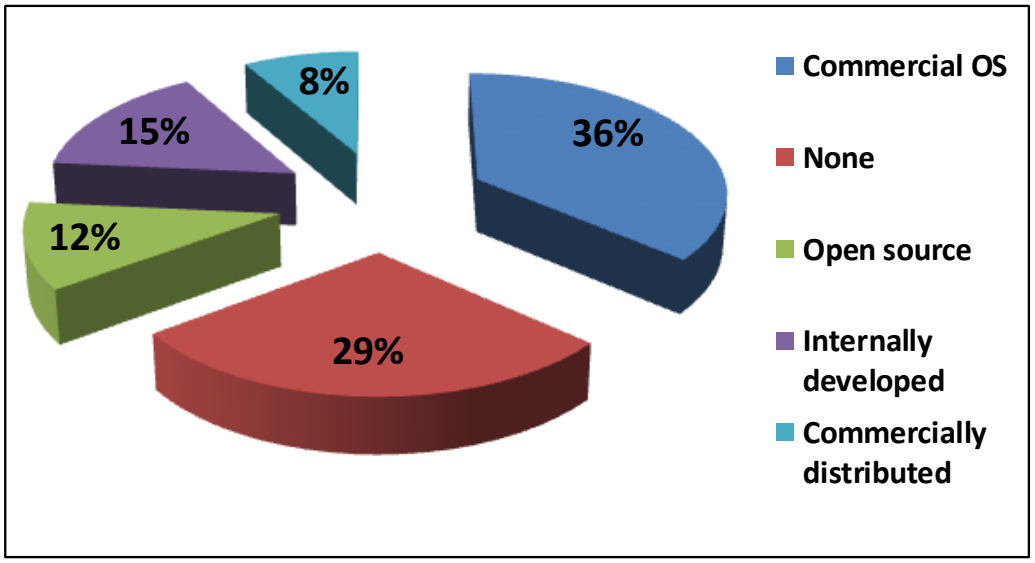


Figure 1.1: The use of operating systems in embedded systems development [111]

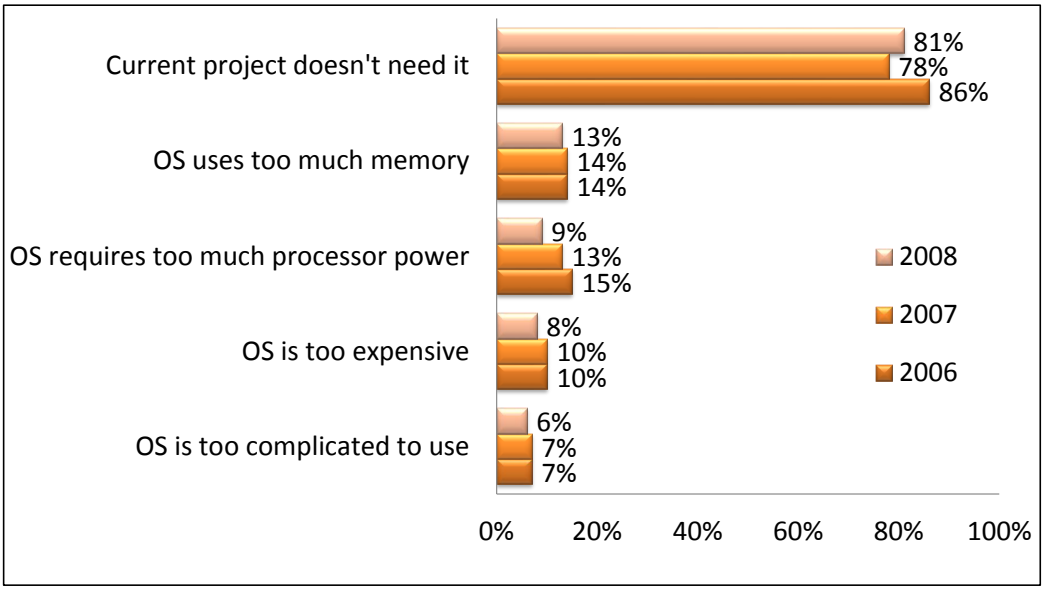


Figure 1.2: The reasons for non-use of operating systems in embedded projects [72]

percentages of embedded OS non-use reasons from a large-scale survey of embedded systems developers from around the world. Power consumption of OS services is one of three important reasons for non-use of OS in embedded systems [72].

In fact, the embedded OS drives the exploitation of the hardware resources

efficiently by offering a wide variety of services such as task management, scheduling, inter-process communication, timer services, I/O operations and memory management. Also, the embedded OS manages the overall power consumption of the embedded system components. It includes many power management policies aiming at keeping components in lower power states, thereby reducing energy consumption.
In this context, the work presented in this dissertation has been carried out at Electronics, Antennas and Telecommunication Laboratory (LEAT), from University of Nice-Sophia Antipolis. This thesis has been conducted under the French national project OPEN-PEOPLE [12] (Open Power and Energy Optimization PLatform and Estimator) gathering several academic and industrial partners (INRIA of Lille, INRIA of Nancy, Lab-STICC (Lorient), IRISA-Cairn (Rennes), THALES Communications (Colombes) and InPixal (Rennes)). The main goal of this project is to provide a complete platform to ease the design of complex systems. This platform should allow rapid power/energy estimation for complex heterogeneous systems, also, it should test different optimizations in order to significantly reduce the power consumption of the system.

1.2 Contributions

Various works have been done on energy consumption characterization and modeling at different abstraction levels in embedded systems design. Many methodologies deal with low level models, and are dedicated to the analysis of hardware components (processor, memory or FPGA), or part of hardware components. With the complexity of embedded systems, these methodologies become not used to analyze the energy consumption. In this thesis, we will focus on approaches intended to estimate the power and energy consumption of a complete embedded system, including its operating system, in order to evaluate the performance and efficiency of low power scheduling policies which are used in various embedded systems. Therefore, the main contribution of this thesis is proposing a methodology for characterizing and modeling the power/energy consumption of an embedded system including software components, the application and operating system, and hardware components. To achieve this goal, we intend to pursue a methodology beginning from power/energy modeling to high level estimation of OS services energy overhead when using lower power scheduling policies:

- We propose an approach of embedded OS power/energy consumption characterization and modeling. Various methods are used to calculate energy and power consumption overheads of a set of three basic services of the embedded

OS: the scheduling, context switch and inter-process communication. Furthermore, the variation of power/energy consumption of the embedded OS services is studied. Also, the impacts of hardware and software parameters like processor frequency and scheduling policy on energy consumption are analyzed. Afterwards, we extract the power and energy consumption mathematical models and laws. The use-case embedded system used is the OMAP3530 EVM board with an OMAP3 processor and Linux 2.6.32 operating system.

- Then, we implement high level models of OS services, software and hardware components taking into account the energy consumption and scheduling requirements. The obtained models will be exploited for calculating OS energy overhead when adapting low power scheduling policies. Also, they will be used for system design exploration and verification of requirements. The architecture modeling language used is AADL. We exploit AADL language functionalities and tools to model the OS services and the software application, the H.264 video decoder application. In addition, the communication between OS services and the applicative tasks has been modeled. Besides, AADL models of OMAP3 processor and the binding of applicative tasks on the hardware platform components have been developed.

- Thereafter, mathematical and AADL models of OS services have been integrated at system level using a multiprocessor scheduling simulator (STORM) in order to evaluate the OS energy overhead when using low power scheduling policies: the DPM and DVFS. In addition, a global approach of models integration is introduced. It is based on three focal concepts: AADL Modeling, code transformation from AADL to STORM and OS services power/energy estimation.

- Finally, we propose a design space exploration methodology and a flow of definition and verification of system requirements. The AADL models of software and hardware components are analyzed quantitatively using the Quantitative Analysis Modeling Language (QAML) and QEML tool. The definition and analysis of system requirements are performed using the (RDAL) language and RDALTE tool. The formal language OCL (Object Constraint Language) is used to describe different constraints and to communicate between AADL and QEML models. Taking into account the system constraints, we propose a design exploration methodology: the first step in this strategy consists of searching the operating point that satisfies the maximum number of system requirements. Once the operating point is checked and validated, the design model can be reviewed and updated. The second step consists of finely reducing the exploration domain by limiting the number of execution units. The

target of third and last step is the allocation of execution resources to each thread once the operating point and processor numbers of our system are predicted and fixed beyond the previous two levels.

1.3 Outline

This dissertation is organized as follows. After the introduction, we present in chapter 2 the research issues associated with power/energy estimation and characterization techniques for processor based embedded systems; at different abstraction levels, from the functional level to the transistor level. The third chapter details the methods used to determine energy and power overheads of a set of three basic services of the embedded OS: scheduling, context switch and inter-process communication. This chapter presents also the extracted OS services energy/power consumption mathematical models and laws. Fourth chapter deals with the AADL modeling of OS services and the software application, the H.264 video decoder application. Also, it presents the AADL models of the communication between OS services and the applicative tasks. The fifth chapter focuses on the integration of AADL and mathematical models in STORM simulator and calculating the OS energy overhead when adapting low power scheduling policies. A flow of definition and verification of system requirements when allocating applicative tasks to the processors is proposed in the sixth chapter. Using a set of languages, RDAL and QAML, various real time and energetic constraints are checked when exploring the design. Finally, the thesis is concluded and perspectives are drawn.

Chapter 2

Background on embedded systems energy consumption characterization, modeling and analysis

Contents

2.1 Power and energy dissipation in embedded systems 7
2.2 Power and energy consumption characterization and estimation in embedded systems 9
2.2.1 Power and energy in electrical circuits 9
2.2.2 Overview of power and energy consumption characterization and estimation 10
2.3 Power and energy consumption estimation of embedded systems at different abstraction levels 11
2.3.1 Estimation and modeling of power/energy consumption at microprocessor abstraction levels 13
2.3.2 Power/energy consumption of hardware components and peripheral devices 20
2.3.3 Characterization of embedded OS power/energy consumption 22
2.4 Conclusion 30

In this chapter, we present the terminology of power and energy dissipation, characterization and estimation in embedded systems. Then, we introduce research efforts and tools related to power and energy consumption estimation of embedded systems at different abstraction levels: the microprocessor, hardware and software abstraction levels.

2.1 Power and energy dissipation in embedded systems

Nowadays, the number of transistors in electronic circuits grows with the development of technology so that each circuit might have millions of transistors packed

inside. Moore's law [54] confirms this huge increase of number of transistors on a chip. According to him, this number doubles approximately every two years, for decades.

Due to the huge number of transistors in electronic devices, different issues related to the overall performance of a system appear prominently. Energy consumption of embedded systems is an important issue that resulted from the huge growth of the number of transistors.

Reducing energy and power dissipation of embedded systems is now a critical challenge for a large number of electronic corporations.

Now, to analyze, characterize and estimate the energy consumption, we examine the CMOS (Complementary Metal-Oxide Semiconductor) circuits that embedded systems consist of.

In CMOS technology-based systems, there are two principle sources of power dissipation: 1) dynamic power dissipation, which arises from the repeated capacitance charge and discharge on the output of the hundreds of millions of gates in modern chips and depends on the processor frequency, and 2) static power dissipation which arises from the electric current that leaks through transistors even when they are turned off.

The power dissipation P in CMOS gates including dynamic and static components is depicted by equation 2.1:

$$P = P_{dynamic} + P_{static} \tag{2.1}$$

Where $P_{dynamic}$ and P_{static} represent respectively the dynamic and static power dissipation.

Also, the total power dissipation in CMOS system can be represented by this expression 2.2:

$$P = P_{SW} + P_{SC} + P_{LK} \tag{2.2}$$

Where, P_{LK} is the static power or leakage power, the remaining terms represent the different parts of dynamic power consumption.

The first term of the equation P_{SW}, represents the switching power dissipation, the major contributor in dynamic power consumption, which is caused by the charging and discharging of gate capacitances when the output changes between high and low levels. During the transition of the output signals, an amount of power P_{SC} is dissipated, the short-circuit power, due to the direct path between the power supply and ground.

The dynamic power dissipation, $P_{dynamic}$, of a CMOS circuit is depicted by an approximate relation given by equation 2.3 which relates the operating frequency F_{op} to the supply voltage V_{op} and the total load capacitance of all gates C_T.

$$P_{dynamic} = \lambda \times C_T \times V_{op}^2 \times F_{op} \tag{2.3}$$

where λ is the activity factor -i.e., the fraction of the circuit that is actively switching.
In a CMOS device including n transistors $\{TR_i, 1 \leq i \leq n\}$, the static power consumption, P_{static}, is calculated as a function of the number of transistors, the leakage current Ilk_i of each transistor TR_i and the supply voltage V_{op}. It is represented by equation 2.4

$$P_{static} = \sum_{1 \leq i \leq n} (Ilk_i \times V_{op}) \tag{2.4}$$

Actually, when not switching, transistors in CMOS circuits lose negligible power, the static power. However, due to the shrink of transistors size, the augmentation of device speed and chip density, the power they consume has increased dramatically. Consequently, the amount of current leakage raises [84].
The leakage of power becomes a significant issue in embedded systems as it reduces the battery service life. To this effect, embedded systems designers propose various techniques aiming at controlling and minimizing the OFF current of CMOS circuits in both standby and active modes of the circuit [49].
In this work, we consider that static power is an important factor that influences the total power consumption of the device and can not be disregarded any further.

2.2 Power and energy consumption characterization and estimation in embedded systems

2.2.1 Power and energy in electrical circuits

Power and energy consumption are important performance metrics for embedded systems. In electrical circuits, the power P is the rate of doing work. It is produced by an electric current I, consisting of a charge of Nc coulombs every Ns seconds, passing through an electric potential difference or voltage V. It is measured in watts (W). The power is given by equation 2.5:

$$P = (Nc/Ns) \times V = I \times V \tag{2.5}$$

Formally, the energy consumed by a system is the amount of power dissipated during a certain period of time. For instance, if a task T occupies a processor during an execution interval of $[a, b]$ then the energy consumed by the processor E_T during this time interval is given by equation 2.6:

$$E_T = \int_a^b P(t)\, dt \tag{2.6}$$

The following section covers the basic terminology that we will use in this thesis dissertation. An overview of power/energy consumption characterization and estimation will be presented. Then, in subsequent sections, we detail the different

abstraction levels of microprocessor based embedded systems. Furthermore, we review the state of the art of power/energy consumption estimation and analysis of microprocessor based embedded systems at different abstraction levels.

2.2.2 Overview of power and energy consumption characterization and estimation

2.2.2.1 Power/energy consumption characterization

Characterizing energy consumption of an embedded system consists in studying the variation of energy consumption of its hardware and software parts. This step aims at determining the energy overhead of different components of the system as a function of various parameters. As a result of the characterization step, mathematical models and laws of the power and energy consumption are extracted : the modeling of energy overhead. The extracted models depend on the parameters varied to characterize the power and energy dissipation. The precision of the models is checked by calculating the error rate which is the difference between the model's values and the measured or estimated values. The energy consumption characterization is based on direct measurements on the platform or on energy estimations.

2.2.2.2 Power/energy consumption estimation

In the design flow, the power estimation is a process allowing the evaluation of the power consumption of an existing design independently of the abstraction level. It aims to check whether power and reliability constraints are verified or not. Estimation step helps to choose the chip parts that ensure a low cost for the embedded systems designers. Depending on the embedded system complexity, functionalities and measurement points, the energy consumption could be estimated based on either power and energy consumption models, deduced from a lower abstraction level, or on simulations using specific simulators or on verification resources in the design flow [23]:

- **Power estimation using simulation:** This kind of power estimation is based on simulations of the embedded system energy consumption. The simulation is proportional to the activity/toggles of the design.

- **Power estimation using mathematical models:** Estimation of energy and power consumption can be made based on mathematical models that describe the dependance of power consumption of the embedded system on certain parameters such as the processor frequency, the memory size, the capacitance, etc.

- **Power estimation using verification resources in the design flow:** Different verification resources, which are a set of design tests built during the verification process of a high-level design, are used to enable power consumption estimation and profiling. Generally, this estimation is specific to a high-level synthesis and power estimation framework.

At an abstraction level $(n-1)$, the estimation process generates power/energy consumption models that can be used to characterize the energy consumption at an abstraction level (n). Figure 2.1 depicts the global flow of power/energy estimation, characterization, modeling and analysis at abstraction levels $(n-1)$ and (n).

2.3 Power and energy consumption estimation of embedded systems at different abstraction levels

The power/energy estimations are centered around two aspects: the power model granularity and the system abstraction level. The first aspect concerns the granularity of the relevant activities on which the power model relies. It covers a large spectrum that starts from a fine-grained level, such as logic gate switching, and stretches out to a coarse-grained level like hardware component events. Fine-grained power estimation, in general, yields a more correlated model with data and handles various technological parameters. On the other hand, coarse-grained power models depend on micro-architectural parameters that cannot be determined easily. Let us highlight that the power estimation accuracy is not altered by the chosen granularity level, however, it depends first on the characterization phase of each activity and second on the computing of the related occurrences while carrying out the application. Even when using coarse-grained activities, the characterization in term of power or energy cost can always be done at a lower level (board measurements, transistor, gate or RTL), and after that, the obtained values can be used at a higher abstraction level. The second aspect of power/energy estimation involves the abstraction level on which the system is described. It starts from the usual Register Transfer Level (RTL) and extends until reaching the algorithmic level. As we go from higher to lower levels, the power evaluation time increases, which is indirectly proportional to the accuracy. The aspects presented above are correlated. Indeed, different power estimation speed/accuracy trade-offs can be achieved according to the power model granularity and the abstraction level from which the relevant activities should be extracted. Figure 2.2 shows the different abstraction levels of microprocessor based embedded systems. The view of the components of microprocessor-based embedded systems is considered as the system-level view. Three main abstraction levels constitute the system level view which are: the software, the microprocessor and the hardware abstraction levels.

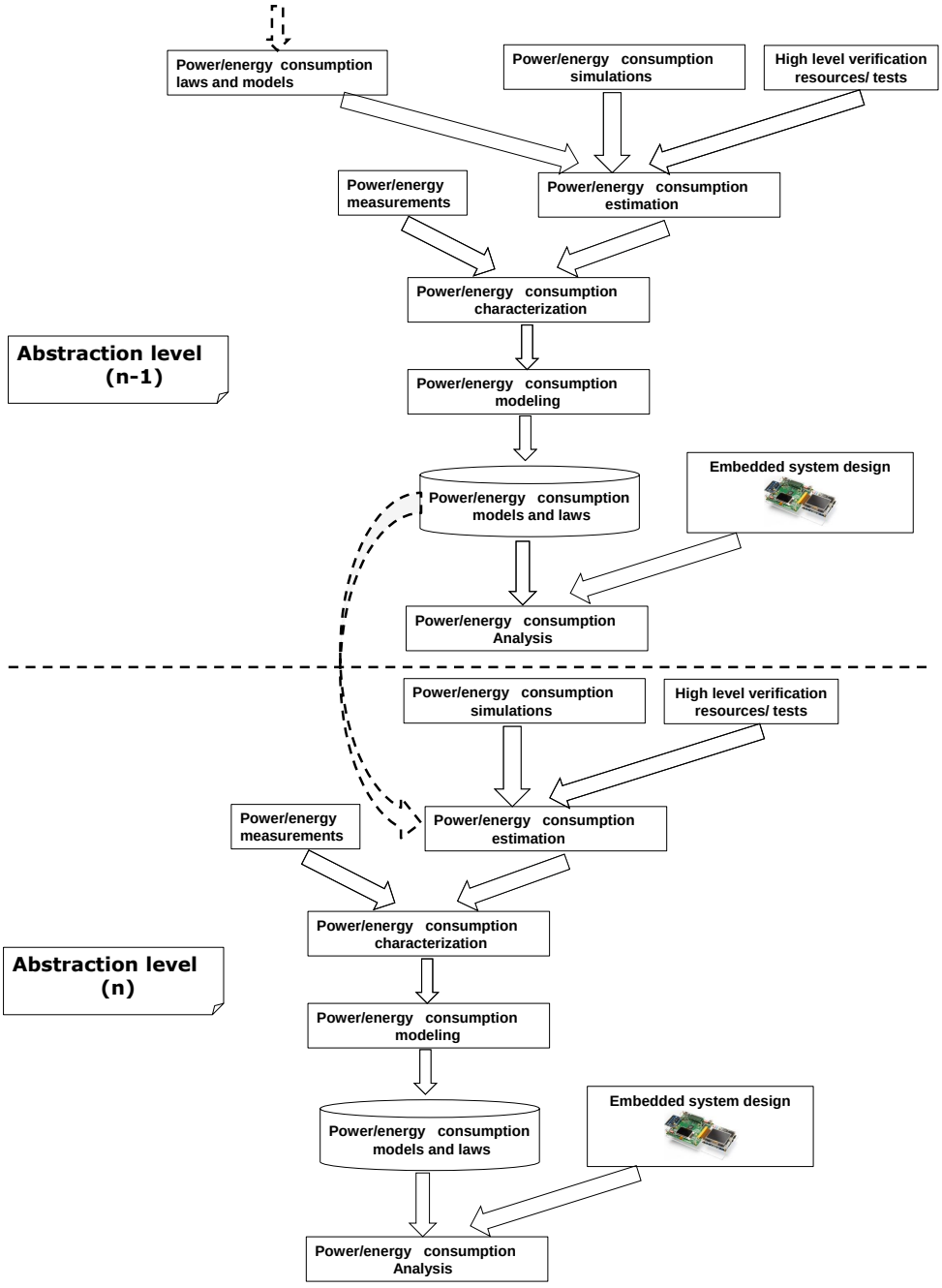


Figure 2.1: Energy/power consumption estimation and characterization flow

The software abstraction levels describe the software part of the system including applications and operating system; from a high level model to the assembly language. The microprocessor abstraction levels extend from the transistor level to the func-

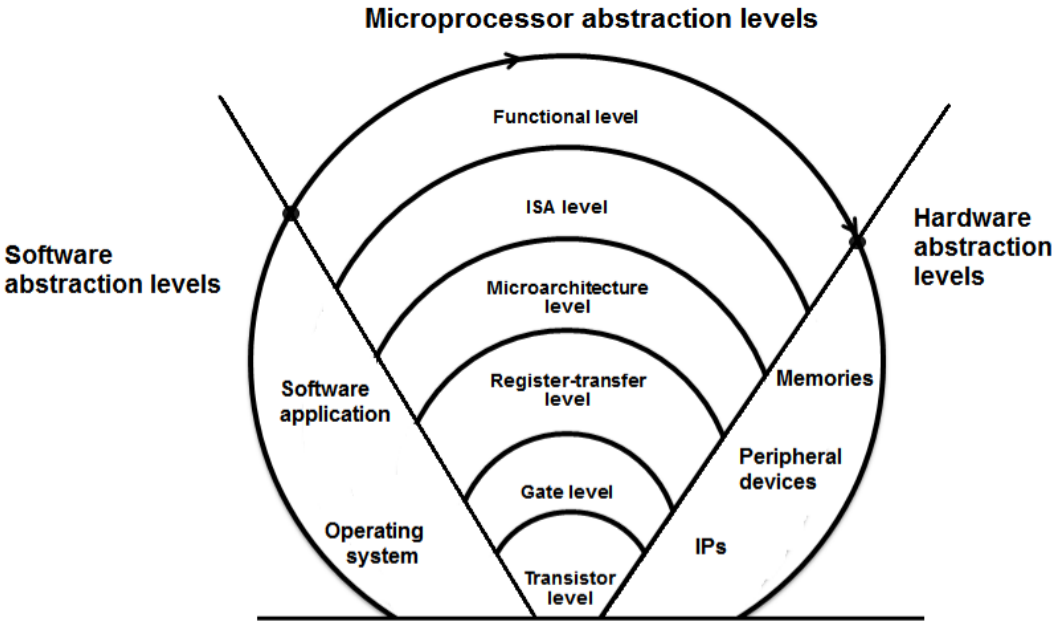


Figure 2.2: Different abstraction levels

tional level. Various research efforts have been devoted to develop methodologies and tools for characterizing, estimating and modeling power/energy consumption at different abstraction levels in embedded system design. In the the remainder of this section, we present an overview of these approaches and tools.

2.3.1 Estimation and modeling of power/energy consumption at microprocessor abstraction levels

In this section, we present related works and tools for different microprocessor abstraction levels: the transistor level, the gate level, the register-transfer level, the micro-architecture level, the instruction set architecture level and the functional level.

- **Functional level:**
  At this level, the processor's architecture is described. This architecture is divided into different functional blocks. Each block represents a unit ensuring a specific functionality such as the memory unit, the processing unit etc. Using simulations or measurements, the power consumption of each block is modeled. The total power consumption is then given as the sum of the power

consumption of each functional block [90]. This division aims at clustering the components that are concurrently activated when a code is running.
In [65], the authors introduce a new instruction level power estimation methodology named the Functional Level Power Analysis (FLPA). In order to estimate the power consumption of hardware platform when running a specific application, the proposed approach is based on identification of functional blocks influencing the processor power consumption. Laurent et al. [65] divide the processor architecture into different functional blocks. Then, they vary algorithmic parameter values which depend on the executed algorithm, and architectural parameters values which depend on the processor hardware characteristics. Finally, they study the variation of processor power consumption as a function of these parameters. Using this methodology, a power estimation tool is developed: the SoftExplorer tool [91]. This tool realizes the suitable tradeoff between the estimation accuracy and time in order to ensure a rapid and reliable feedback to the designer.
In [92], the SoftExplorer tool is used to estimate the power/energy consumption of an algorithm directly from the C program. Also, it is used to optimize the power/energy consumption of an application. The authors perform a functional-level power analysis to extract the different power models and describe how to perform the best data mapping for an algorithm. To validate the proposed methodology, various processors have been used such as the (ARM7), the low-power (C55) and the VLIW (C62) processors. Furthermore, very important phenomena like pipeline stalls, cache misses, and memory accesses are taken into account.

- **Instruction set architecture level:**
  The Instruction-Set-Architecture abstraction level is related to programming aspects. It describes the processor's addressing modes and registers. Also, it includes instructions that can be executed on the processor, the set of native commands and machine language instructions that specify the operation to be performed. During the instruction set architecture power estimation, a power consumption cost is assigned to each individual instruction, by considering pipeline stalls and cache misses.
  Instead of analyzing the hardware components behavior, such as the number of memory access and power overhead of transistors, several studies have proposed instruction-level approaches to model the energy consumption.
  The work done in [33] relies on instruction-level energy estimation for VLIW (Very Long Instruction Word) processors. Power consumption have been modeled for various components of the system: the core, the register file, the instruction and data caches.

This work shows the reduction of the complexity of the energy model for VLIW cores, while preserving a good level of accuracy. Then, the authors proposed an estimation engine that provides power consumption estimates for software running on a given hardware architecture by interpreting an executable program and simulating and profiling the effects of each instruction on the main components of the architectural state of the system. To obtain an overall hardware/software power optimization for VLIW embedded systems, the authors use various instruction-level techniques such as operation clustering which consists in grouping in the same cluster the operations with energy cost values close to each other.

In order to estimate the energy consumption of a given program under different cores and to find the energy-optimal number of cores used for execution, the authors in [115] proposed an instruction-level methodology consisting in predicting the energy consumption. The mechanism of prediction is achieved using an output of initial programs compilation called Parallel Thread Execution (PTX) codes [8], a pseudo-assembly language used in NVIDIA's CUDA [6] programming environment. The output of the proposed approach is the estimated energy consumption under different number of active stream multiprocessors. Tests have been carried on several NVIDIA CUDA benchmarks. Wang et al. [115] assign to each type of PTX instructions an energy overhead. As showed in equation 2.7, the energy consumption within one thread is the sum of products of unit energy consumption of one type of PTX instruction and the number of instructions of that type.

$$E_{thread} = \sum_{1 \leq i \leq n} (e_i \times n_i) + o_1 \tag{2.7}$$

where E_{thread}, e_i and n_i represent respectively the energy consumed by one thread, the energy consumption of a certain type of PTX instruction and the number of instruction of type i . The parameter o_1 is the energy consumed when the thread is created.

- **Micro-architecture level:**
  In order to implement the set of instructions, the interconnections of different parts of the processor and the communication between the micro-architectural components of the machine, such as processor registers and caches, are described at the micro-architecture abstraction level.
  Hidaji et al. [57] are interested in power estimation and optimization at micro-architectural abstraction level. To optimize power consumption, different techniques have been used, such as the Clock-gating technique that is based on stopping the clock of some parts of the design when these parts are idle and therefore reducing the switching power, and the common-case technique by

spotting the most common operation conditions and optimize their switching power. Also, Hidaji et al. achieve a memory optimization using the common-case technique.

In [62], Kim et al. proposed micro-architectural level power modeling methodologies approach for deep sub-micron microprocessors. In order to determine the execution time and circuit specific power consumption, the authors included detailed micro-architectural and circuit models in their approach. They modeled power consumption of different micro-architectural components, such as transistor capacitance components, by switching events with an embedded cycle-based logic simulator, execution units and memory accesses etc. Also, they introduced an accurate micro-architectural event modeling methodology to give a cycle-accurate power estimation of long-latency and multi-cycle operations such as external I/O access. The proposed technique combines simplified circuit-level capacitance extraction and cycle-based logic simulation embedded into a micro-architectural level simulator: SimpleScalar [25].

A new technique for processor power optimization at micro-architectural level, called micro-architectural power analysis (MPA) for microprocessors, is presented in [37]. The simulation environment used in this work is the Simplescalar microprocessor simulator. The proposed technique consists of three mechanisms: dynamic, static and multivariate power analysis. Based on architectural simulations and dynamic power models, the dynamic power analysis is performed by analyzing the power consumption behavior. The authors integrate power monitors, which are parts of power models, into each functional block within the architectural performance simulator to determine the power consumption of each block. In the static micro-architecture power analysis step, the power consumption of a microprocessor under different workloads is estimated. This analysis permits the estimation of full-chip and each functional block power consumption when running different applications such as office applications, games and multimedia applications, scientific computational applications etc.

Using the results of the dynamic and static micro-architectural power analysis, the multivariate power/performance analysis step identifies the possible power reduction targets within complex microprocessor architectures. Cai et al. [37] study the effectiveness of the proposed micro-architecture circuit implementation by evaluating the full-chip power consumption, the functional block power consumption, the average performance and the interfaced functional block impacts. For instance, they compare the power reduction between two different implementations A and B: the full-chip and functional block power reduction rates are respectively 2% and 10% when using the circuit implementation A, whereas, when the other circuit implementation B is used, these rates are

respectively 3% and 20%.

- **Register-transfer level:**
  In digital circuits, the register-transfer level (RTL) abstraction level describes the mechanism for exchanging data between hardware registers using digital signals. It also details the logic functions of these signals. To estimate accurately the power consumption of a circuit at the RTL abstraction level, the hardware/software designers need three entries: a design description using a specific hardware description language such as Very-high-speed integrated circuits Hardware Description Language (VHDL) [15], a trace of RTL simulation using a standard file format, such as the Value Change Dump (VCD) format, and power/energy characterization libraries.
  In [24], Ahuja et al. present a system-level power estimation methodology, which is based on a high-level synthesis framework and supports sufficiently accurate power estimation of hardware designs at the system level.
  To provide a reasonable power estimation while remaining at high level, the authors propose a methodology using register-transfer level probabilistic power estimation technique controlled by the system-level simulation. As showed in figure 2.3, the methodology is divided into different steps: first, Ahuja et al. convert the system-level model of the design to an equivalent cycle-accurate RTL model using Esterel Studio tool [11] in oreder to synthesize the high-level models to RTL implementations. Then, they simulate the high-level model and generate its VCD file. After that, they apply an algorithm to the VCD file generated in order to extract the activities associated with each signal of the RTL design, such as the number of simulation ticks for which the variable value remains unchanged, from the system level simulation dump. Finally, they perform the mapping of system-level variables to RTL signals and use the algorithm outputs to find the activity information of the remaining signals, to generate the power models and to analyze the power consumption using RTL power estimator: the "PowerTheater".

- **Gate level:**
  The gate level describes the flip flops wire-connected to the logic-gates (such as NOT, AND, NOR, etc.). The estimation of power consumption at this level is achieved using a specific libraries of logic gates. Each library provides the different elements allowing the power characterization of different gates. The embedded system designers can calculate the currents in the various logic gates and thus monitor the power consumption of the circuit.
  In [95], energy consumption modeling is done from a gate-level description. The authors associate the energy cost with the occurrence of certain architectural events such as the ALU events which represent a logic instruction

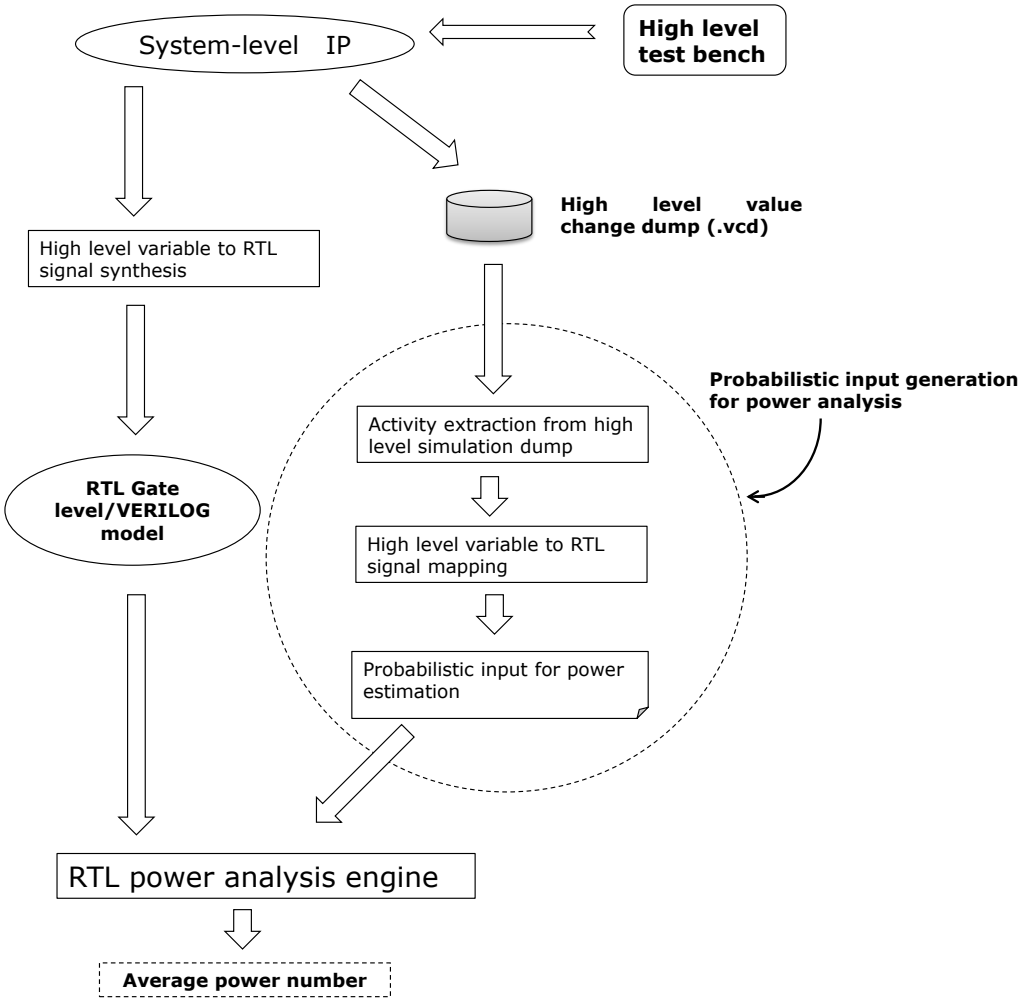


Figure 2.3: Ahuja et al.'s RTL power estimation flow [24]

(addition, substraction etc.). These events, which are modeled for timing and energy estimates, are computed simply by some additional counting. In fact, the proposed model assumes that the total energy consumed by a processor is the sum of its idle energy consumption E_{idle} and the energies consumed by the different events executed by the pipeline, execution units and caches. The total energy consumption of an application E_{total} is depicted by equation 2.8:

$$E_{total} = E_{idle} + \sum_{i} (e_i \times n_i) \tag{2.8}$$

Where i and e_i represent respectively the set of events and their energies. The parameter n_i denotes the number of executed events of event type i.
To validate their approach, the authors decided to model the power consump-

tion of a specific microprocessor: the PowerPC 405GP core.
Tran et al. [109] propose a methodology to estimate the power consumption of digital CMOS VLSI chips. They divide the chip into five parts: the logic circuit, the memory, the local and intermediate interconnections and global buses, the clock distribution and the I/O drivers. After that, they characterize the power overhead of each chip part. To estimate the power consumption at gate level, Tran et al. implement an estimation tool using the C language. The proposed tool is used to count the number of gates (such as NOR, XOR, inverter, multiplexer, etc.) needed to implement the design description. Then, this tool calculates the gates number of each arithmetic and logic operation. The gates count of each single instruction is calculated independently and all single results are added to the global statement gate-count result.

- **Transistor level:**
The transistor level is the lowest abstraction level. It is based on modeling the behavior of the basic electrical elements such as transistors, resistors, capacitors... etc, and describing the interconnection between these primitive electrical elements. The behavior description is generally performed by using equations or specific diagrams.
In [27], Basmadjian et al. perform a transistor level power estimation and characterization of idle servers. The authors provide power models for multi-core processors, hard disks, memories, power supply units and fans. Basmadjian et al. identify the relevant energy-related attributes allowing to build the basis for the power consumption prediction models. Since the power consumption of each core depends upon its number of transistors, the authors consider the power consumption of the processor core as the sum of power consumption of its transistors. The power consumption $P_{j,i}$ of j^{th} transistor, inside the i^{th} processor's core, is given by equation 2.9.

$$P_{j,i} = I_{j,i} \times V_{j,i} \tag{2.9}$$

where $I_{j,i}$ and $V_{j,i}$ represent respectively the current and voltage of the j^{th} transistor of the i^{th} core.
The authors study the impact of frequency for a given voltage on the power consumption of different server's components. They set up machines having various hardware characteristics, such as the processor's type: Intel and AMD processors, the number of cores: dual-/ quad-/ hexa-core processors, different memory modules: DDR2 and DDR2, as well as various energy-saving mechanisms (e.g. Intel SpeedStep, AMD Cool'n'Quiet).
A new approach for simulating a transistor-level design with a VHDL test-bench was adopted in Singh et al.'s work [101]. The proposed test-bench, implemented using the Mentor Graphics digital design tool suite [2], reads

the transistor level design's outputs and supplies the inputs accordingly. The proposed method calculates automatically the power and energy consumption and performs automated testing of functional correctness. Singh et al. apply this approach to specific circuits, the NULL Convention Logic (NCL) circuits [50], and their transistor-level designs were successfully simulated using self-checking exhaustive VHDL test-benches.

In [100], Shiue et al. estimate the power consumption at transistor level using a new analytical equation derived from a particular model, based on the physical law MOSFET [34] models and BSIM3v3 manual [1], having thus a simple mathematical form and ensuring a high degree of accuracy for the power estimation of CMOS circuits. The proposed analytical equation model was validated on their own benchmark example and shows 2.72% error in average.

Based on the internal capacitance switching and discharging currents of such circuits, the authors present in [87] an accurate analytical expressions to compute the dissipated energy and the propagation delay of CMOS gates. They use a good metric to evaluate a design called the energy delay product (EDP). The obtained results show that the position of the switching transistor on the overall gate delay can lead to a 20% of delay variation. Also, this study concluded that short-circuit current of the output inverter optimizes the gate's energy consumption.

Next section details the related works and tools to power/consumption estimation at hardware abstraction level.

2.3.2 Power/energy consumption of hardware components and peripheral devices

In embedded systems, hardware devices can consume an important amounts of energy. In this paragraph, we report research studies that are interested in energy consumption estimation and modeling of hardware platform components.

Celebican et al.'s work [39] focuses on the energy estimation of peripheral devices in embedded systems. To identify the highest power consuming routines and components, Celebican et al. implement a cycle-accurate simulator and profiling tool for energy consumption of hardware platform devices. They modeled with analytical equations the energy overhead of I/O controller and the audio module. Also, they define different energy modes for each peripheral device. The energy value of each peripheral in each mode is calculated. The proposed approach is tested with two types of communication protocols, the polling and the interrupt based communication protocols. Also, they compare the energy consumption of various hardware modules when using the polling communication protocol and the Direct Memory

Access (DMA) functionality that enables direct communication between memory and peripherals. For experimental framework, the Linux based development board SmartBadge IV [3] is used as an embedded system with different hardware components, such as the processor, the memory, an audio interface, a speaker and a headphone. Simulation results show that peripheral devices can consume from 50% to 55% of the total system energy consumption. Furthermore, MP3 decoder is used as use-case application to demonstrate that the profiling tool reduces the total application energy consumption by 44% when optimizing the audio driver's energy overhead.

Konstantakos et al. present in [64] a new methodology for modeling the energy consumption of various hardware devices. As showed in equation 2.10, they consider that the energy consumed by an embedded system when executing a software application is mainly the sum of energy overheads of a micro-controller, a RAM memory and an analog-to-digital A/D converter.

$$E_{System} = E_{\mu Controller} + E_{RAM} + E_{ADC} \tag{2.10}$$

where $E_{\mu Controller}$, E_{RAM} and E_{ADC} represent respectively the energy consumed by the micro-controller while executing a program, the memory and the A/D converter. The energy of each peripheral device component has been modeled and analyzed based on separate physical measurements of the drawn current. For instance, to characterize the energy consumption of the RAM memory, the authors are interested in studying the energy overhead of each read/write access routines. Also, the standby energy consumed due to the steady-state current flow through memory cells and the energy dissipated during each memory row refresh are considered. Based on measurements, an average value of energy consumption for each read and write access routines and refresh operations is calculated. In the proposed memory energy model, Konstantakos et al. take into account the energy leakage due to the steady-state current, this energy is calculated as the integral of power for the complete period of the test-program execution, as formulated below:

$$\begin{aligned} E_{RAM} = {} & c1 \times n_read_accesses + c2 \times n_write_accesses \\ & + c3 \times n_refreshes + \int_{executionperiod} P_{Standby} \end{aligned} \tag{2.11}$$

where c1, c2, c3, $n_read_accesses$, $n_write_accesses$ and $n_refreshes$ represent respectively the average energy for read, write and refresh operations, the number of read accesses, write accesses and memory refreshes.

The software abstraction levels include mainly the operating system (OS) and the software application. The hardware/software designers aim at estimating the power consumption of the whole system at different abstraction levels, including the soft-

ware level. For this reason, determining and estimating the power/energy cost of the OS are highly required. In the next section, we present various research works related to the characterization, estimation and modeling of power/energy consumption of embedded OS.

2.3.3 Characterization of embedded OS power/energy consumption

In order to characterize energy and power overhead of embedded OS, several studies have proposed evaluating its energy consumption at different abstraction levels.
Dick et al. [45] analyze the power consumption of the μCOS operating system which is running several embedded applications on a *Fujitsu SPARClite* processor based embedded system. This study is the first work that characterizes the power consumption of an OS. The authors developed a general framework to measure the power consumed by the application and operating system routines. They present quantitative results for energy and time consumed by various operating system routines, such as semaphores, task control, synchronization, and timer management. Also, Dick et al. show that power/energy characterization and analysis of OS services help to optimize and reduce the power consumption of embedded system's software layer. This study demonstrates that the OS functions have an important impact on the total energy consumption of an embedded system. This impact depends on the complexity of the applications. This work represents only an analysis of operating system power consumption. Dick et al. did not determine power/energy consumption models and laws.
Tao Li et al. introduce in [66] a routine level power model of OS tasks. As depicted by equation 2.12, the energy consumed by the OS services E_{OS} is considered as the sum of the energies consumed by different OS routines.

$$E_{OS} = \sum_{Alli} (P_{OS_routine,i} \times T_{OS_routine,i}) \tag{2.12}$$

where $P_{OS_routine,i}$ and $T_{OS_routine,i}$ represent respectively the power and execution time of the i^{th} OS routine invocation.
This work evaluates the power characteristics of these OS routines and extract power consumption models. The authors show that OS routines power consumption depends on the test benchmarks. Interestingly, they observe that this power is strongly correlated with OS performance and the Instruction per cycle (IPC) metric. This metric is exploited for power/energy characterization of embedded OS services. When validating the approach to track the OS routines energy overhead, the authors found that the error rate per routine estimation is less than 6%. Low power techniques such as dynamic voltage scaling are not applied to the OS code and not considered in this work.

Acquaviva et al. propose in [22] a new methodology to characterize the OS energy overhead. They measure the energy consumption of the *eCos* Real Time Operating System running on a prototype wearable computer, HP's *SmartBadgeIII*. Then, they study the energy impact of the RTOS both at the kernel and at the I/O driver level and determine the key parameters affecting the energy consumption. This work studies the relation between the power and performance of the OS services and the CPU clock frequency. Acquaviva et al. perform an analysis but they do not model the energy consumption of the OS services and drivers.

In [105], Tan et al. model the OS energy consumption at the kernel level. They classify the energy into two groups: the explicit energy which is related directly to the OS primitives and the implicit energy resulting from the running of the OS engine. The authors explain their approaches to measure these classes of energy and they propose energy consumption macro models. Then, Tan et al. validate their methodology on two embedded OSs, *µCOS* and Linux OS. However, the scope of the proposed work in [105] is limited in some ways as it targets OS's running on a single processor. Also, the authors do not consider the I/O drivers in the proposed energy consumption model.

In [28], Baynes et al. describe their simulation environment, *Simbed*, which evaluates the performance and energy consumption of the real time operating system (RTOS) and embedded applications. The authors compare three different RTOS's: *µCOS*, *Echidna* and *NOS*. They found that the OS energy overhead depends on the applications: it is so high for the lightweight applications and diminishes for more compute-intensive applications. Nevertheless, Baynes et al. perform high level energy simulations to extract power/energy models. These models are not realistic because they are not deduced from measurements on actual hardware platform. Also, the energy consumption of OS services compared with the total application energy consumption was not calculated.

Guo et al. [55] introduce a novel approach using hopfield neural network to solve the problem of RTOS power partitioning; they aim at optimally allocating the RTOS's behavior to the hardware/software system. They define a new energy function for this kind of neural network and some considerations on the state updating rule. The obtained simulation results show that the proposed method can perform energy saving up to 60%. This work does not consider energy macro-modeling and RTOS services.

Zhao et al. [117] propose a new approach to estimate and optimize the energy consumption of the embedded OS and the applications at a fine-grained level. As showed in figure 2.4, the proposed estimation framework consists of three major components: a full-system instruction level simulator to execute the OS and applications; a micro-architectural power simulator to estimate cycle-accurate

power dissipation of instructions and a software energy analyzer to integrate multiple-granularity software energy consumption. The entries of the proposed methodology are an executable binary OS kernel image file and a root file system involving user-level test programs. The output is the energy consumption of OS routines. The image *initrdi.img* (initial ramdisk) describes the load of root file system into memory in the boot process of the Linux kernel. The authors apply the *readelf* command to the executable image file, generated from the compilation of OS source code, and extract an OS symbol information table. The full system instruction simulator simulates functionalities of microprocessor and hardware devices. It generates instruction and address traces and sends it to the micro-architectural power simulator, through a message queue, in order to simulate operations of micro-architectural components of pipelines and memory access. During instructions execution in the pipeline, the simulator calculates the power consumption of micro-architectural components. It sends cycle-accurate power consumption of instructions and corresponding instruction addresses to the software energy analyzer.
The software energy analyzer treats a run-time operating system as a set of logical units consisting of atomic functions, routines, services and execution paths. It builds run-time function call tree on the fly by analyzing instruction-address sequences and symbol information of OS. Then, it calculates multiple-granularity software energy consumption of OS based on software energy estimation-model. Zhao et al. implement their approach, using an *Intel Strong-Arm* architecture running embedded Linux 2.4.18. They show that energy consumption of the embedded OS and the software application could be characterized and optimized.

Fournel et al. [53] present a performance and energy consumption simulator for embedded system executing an application code. This work allows designers to get fast performance and consumption estimations without deploying software on target hardware, while being independent of any compilation tools or software components such as network protocols or operating systems.
Fei et al. [51] are interested in reducing the energy consumption of the operating system-driven multi-process embedded software programs by transforming its source code. They minimize the energy consumed when running OS functions and services. The authors propose four types of transformations, namely process-level concurrency management, message vectorization, computation migration and inter-process communication (IPC) mechanism selection. The authors evaluate the applicability of theses techniques in the context of an embedded system containing an Intel StrongARM processor and embedded Linux OS. They manage process-level concurrency through process merging to save context switch overhead and IPCs. They modify the process interface by vectorizing the communications between processes and selecting an energy-efficient IPC mechanism. This work attempts to

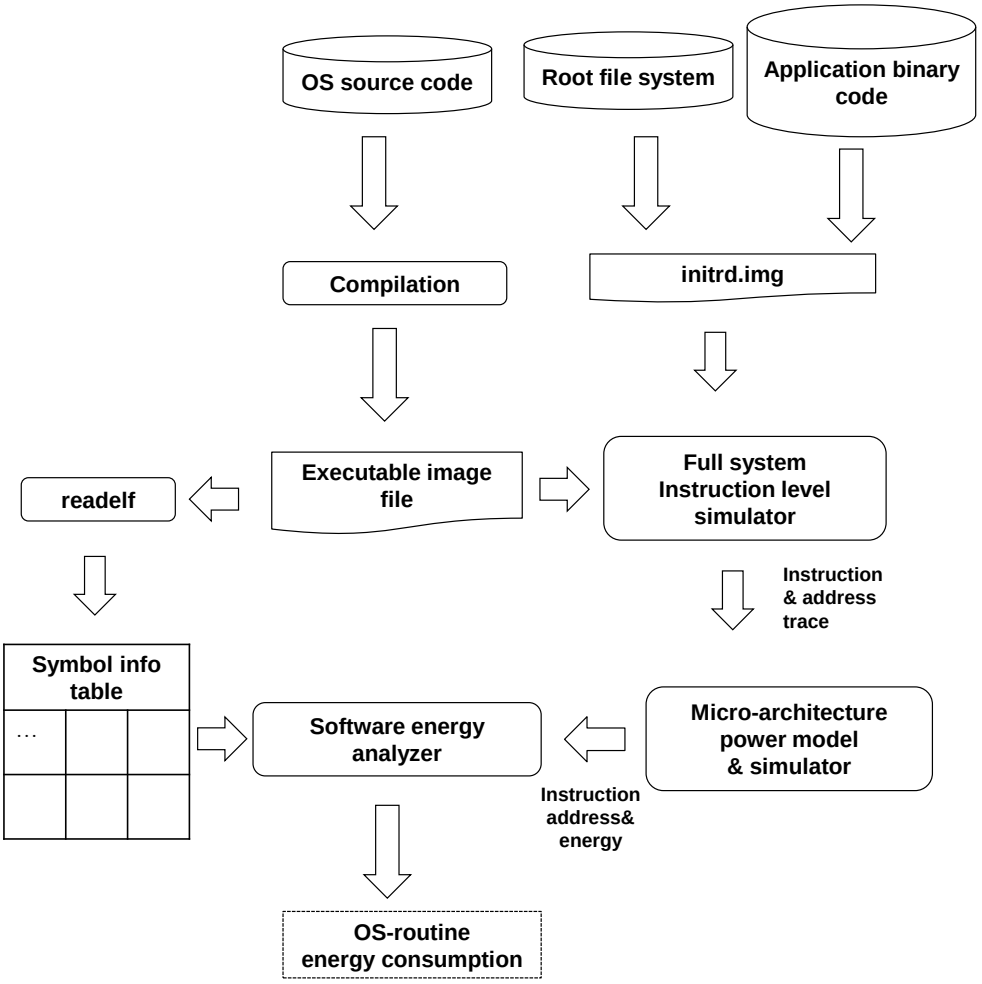


Figure 2.4: Overview of Zhao et al. OS routines energy estimation approach [117]

relocate computations from one process to another so as to reduce the number and data volume of IPCs. These transformations provide complementary optimization strategies to traditional compiler optimizations for energy savings.

Dhouib et al. [44] propose a multi-layer approach to estimate the energy consumption of embedded OS. The authors start by estimating energy and power consumption of standalone tasks. Then, they add energy overheads of OS services which are timer interrupts, inter-process communication and peripheral device accesses. They validate the multi-layer approach by estimating the energy consumption of an M-JPEG encoder running on linux 2.6 and deployed on a XUP Virtex-II pro development board. Low power scheduling policies are not considered in this work.

Brandolese et al. [35] introduce an approach to characterize the OS for embedded

applications. The methodology is divided into two phases: measurements and modeling. The methodology is based on the opportunities offered by SoC hardware/software architectures (Xilinx/PowerPc and Altera/ARM). The main benefits of this work is respecting the characteristics of different OSs and microprocessors, the simplification of the measurement setup and the coverage of system calls of commercial and open source OSs. The obtained results constitute a sound starting point for a more complete analysis of software energy characteristics, both for estimation and optimization purposes and allow covering the whole spectrum from source-level down to system calls.
In [74], Nellans et al. consider two approaches to manage the execution of the embedded OS instructions. They are interested in the boost of performance afforded by reducing OS-user interference within the cache. The authors propose an adaptive off-load policy based on behavior profiling and syscall run-length prediction. They introduce a cache within a core to cache a subset of OS references and consider several design options for it, including various block placement policies, bank predictors, and sequential/parallel look-ups.

Kang et al. [61] present a new approach to characterize the energy consumption of individual OS functions in the $\mu C/OS-II$ real time kernel running on an ARM7TDMI-based embedded system. To measure the energy consumption of different components of the hardware platform such as the CPU, the cache, the memory and the bus, Kang et al. use the Seoul National University energy scanner (SES) [99] as energy measurement tool. Because the SES tool was not designed to attribute energy consumption to the $\mu C/OS-II$ kernel functions, the authors modify its structure slightly to measure the energy consumption of each kernel function. In order to save the OS energy consumption, the authors improve the utilization of the cache memory using the cache locking mechanism: in a first step, to determine the function to lock into the cache, they determine the ratio of the energy consumption which is the percentage of the total OS energy consumption and invocation frequency of $\mu C/OS-II$ kernel functions, such as $OSSched()$ and $OSTaskChangePrio()$ functions. Then, they lock frequently used OS routines into the cache, such as switching and timer interrupts, and rearrange the code to avoid cache contention between these routines. To handle cache locking from an application, the authors add a new layer, the "energy aware" (EA) layer, where the application programming interface (API) and the cache management modified functions were implemented. To lock and unlock the instruction or data cache, the cache management functions were implemented in the hardware abstraction layer (HAL). As depicted in figure 2.5, using the EA API functions, software application can lock each OS kernel function in the cache. The (EA) manager layer sends the function addresses to be locked to the memory simulator, where the requests are handled.

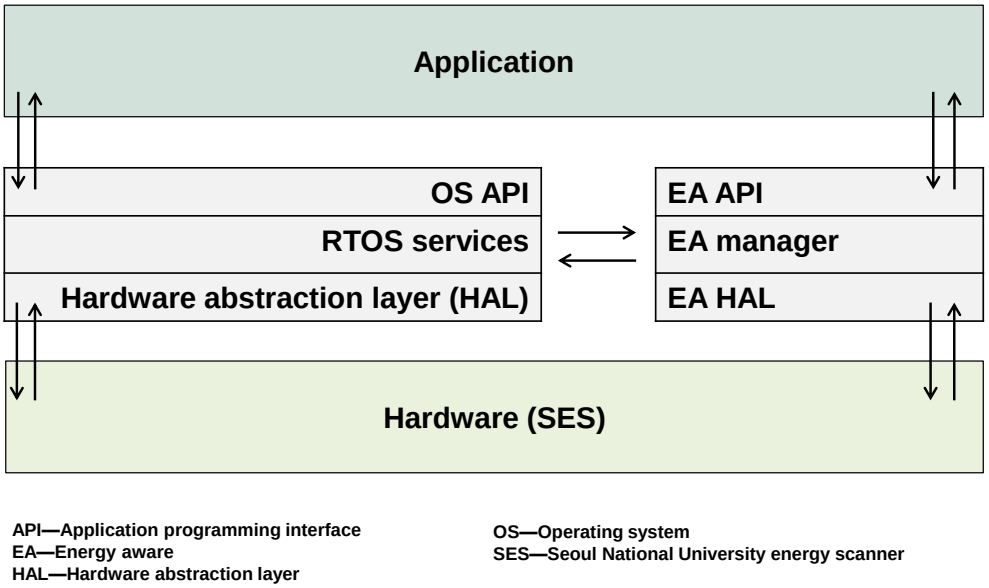


Figure 2.5: Implementation of API and cache management functions [61]

Experimental results show that total energy savings can increase to 5.9%.

In [56], Haukilahti characterizes the energy consumption of a RTOS hardware accelerator called RTU. As depicted by figure 2.6, the flow of the proposed method is presented. The synthesis and power optimization of RTU is performed using the *Synopsys Design Compiler* tool. The author generates a gate-level netlist after the synthesis step. Then, he simulates the generated netlist using an RTL-level simulator called *Modelsim*. To determine the energy consumption of different system calls, such as *thread_create*, *semaphore_create*, and *task_switch*, Haukilahti uses benchmarks that generate one switching activity file for each system call performed by the RTU. Also, the execution times are recorded and the power consumption of each call is estimated using a power analysis tool called *Synopsys Design Power*. The tool performs power estimation at gate-level for each switching activity file. The power optimization process is achieved using a power optimization tool called *Power Compiler* and the switching activity files. The simulations show that the power consumption of the RTU is almost independent of what action it performs. The variation from the average power consumption is less than 4 percent. However, in this work, the energy consumption of OS calls was not modeled and the variation of power consumption as a function of different software or hardware parameters was not studied.

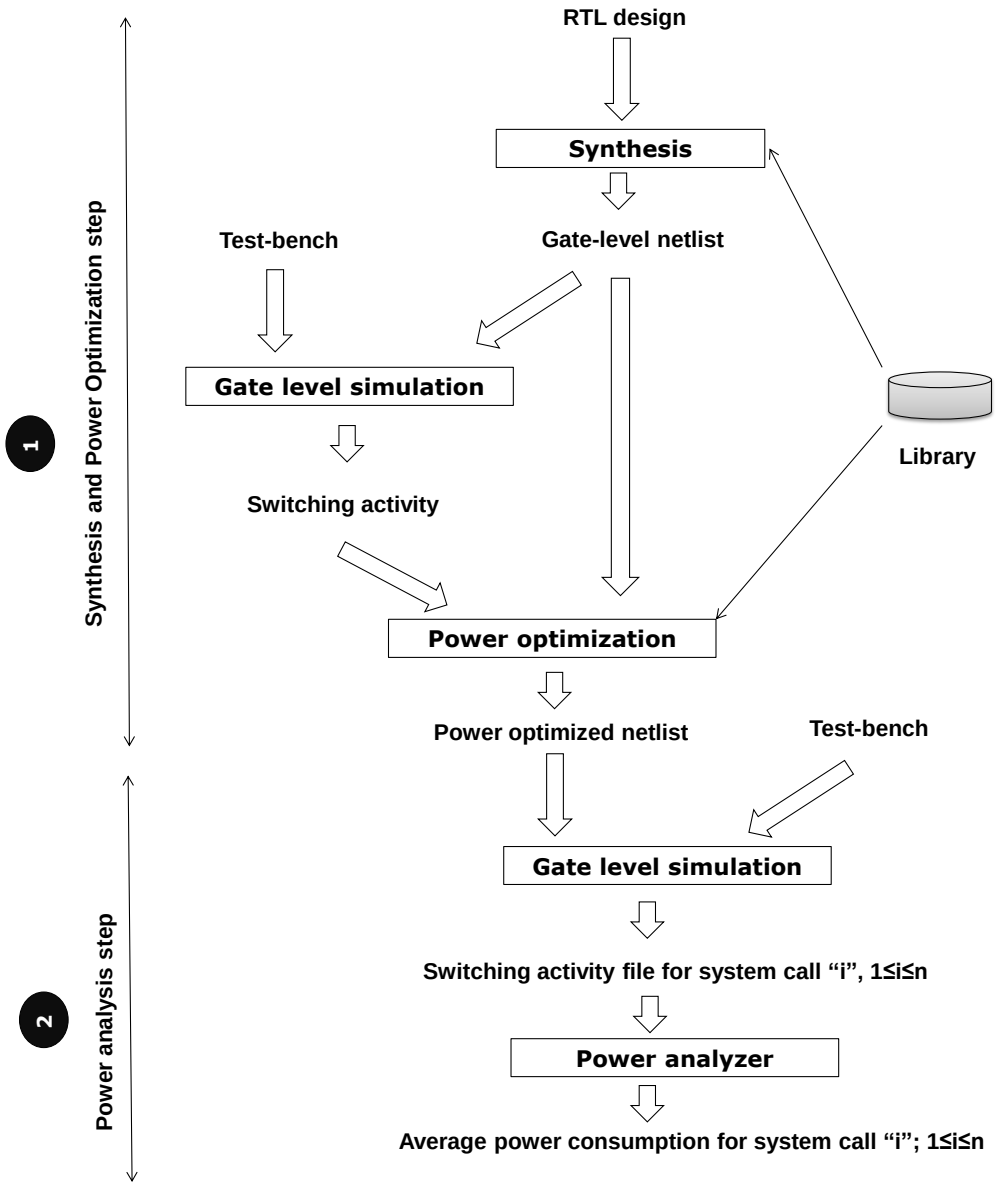


Figure 2.6: Power synthesis, optimization and analysis methodology [56]

In [81], Penolazzi et al. present an accurate and fast methodology to estimate the energy consumption and performance overhead of Real-Time Operating Systems (RTOS). First, they characterize, using RTEMS OS for the Leon3-based SoC, the energy consumption of RTOS main functionalities in terms of typical number of execution cycles and power consumption. The characterization process consists of five main steps:

- **1-Hardware platform synthesis:**

At this step, the authors compose a representative SoC with basic hardware components allowing the binding of RTOS. They synthesize the platform by describing the wire delays and parasitics at gate-level.

- **2-Association of RTOS routines with memory addresses:**
  The authors compile and load the RTOS into the system. Then, they extract a memory dump of the RTOS and its tasks. This dump allows to associate any routine name with its memory addresses.

- **3-Gate level system execution:**
  At this level, the application tasks are executed at gate level and the RTOS calls are activated for a long period of time to ensure a high accuracy of energy characterization.

- **4-Generation of VCD file:**
  This step consists of generating an execution trace file and a VCD (Value Change Dump) file from the gate-level simulation. These files contain data allowing the power consumption estimation such as the execution time of RTOS instructions and the switching activity.

- **5-Determination of execution cycles number and power consumption:**
  This last step targets characterizing the energy consumption of RTOS routines using the memory dump, the execution trace and the VCD files. To extract the average number of cycles and calculate the power consumption of RTOS routines, Penolazzi et al. develop and use a C-based script, called RTOS Modeler. This script allows the energy characterization of RTOS routines or sequences of routines without any intervention from the user.

Then, after characterizing the RTOS activity, the authors propose an algorithm to predict how many times the OS calls get triggered during the OS execution in order to estimate the total OS overhead. The authors compare the effectiveness of their approach with other methods. In fact, They are interested in comparing the accuracy with gate level simulation method and the speed with Transaction-Level Modeling (TLM) approach. For experimental results, the chosen applications are the image compression codec JPEG2000 and the video compression codec H264. Applications and data have been combined in different ways to show some possible use-case scenarios, where two applications always run concurrently on top of the RTEMS OS. Penolazzi et al. show that their approach could achieve an important mean speedup (36X) compared to TLM. But when estimating the energy consumption, they lose 12% of the gate-level accuracy. This work considers only the scheduling and clock tick interruption routines. The energy consumed by other

services of the RTOS, such as the inter-process communication, was not studied. Also, in this work, the power/energy consumption was not modeled.

2.4 Conclusion

In this chapter, we have described the research issues associated with power/energy estimation and characterization techniques for processor based embedded systems, at different abstraction levels, from the functional level to the transistor level.
Embedded operating system is integrated to handle applications upon hardware architectures. Research studies show that the OS not only steals a significant portion of the machine cycles but it can also consume a large part of embedded system's total energy. Therefore, energy and power estimation of operating systems constitutes a challenge for embedded system designers.
In more recent works, characterization of low power OS was not considered. It is not mentioned what are the processor capabilities and which low power policy is used. Also, some works did not model the energy consumption of OS services. In the sequel of this dissertation, we will address all these topics. The next chapter introduces a flow of OS energy characterization. We first study the variation of the energy and power consumption of the embedded OS services. We detail the methods used to determine energy and power overheads of a set of embedded OS basic services: scheduling, context switch and inter-process communication. We analyze the impact of hardware and software parameters like processor frequency and scheduling policy on the energy consumption and we deduce models and laws of the power and energy consumption.

Chapter 3

Characterization and analysis of embedded OS services energy consumption

Contents

3.1 Introduction . . . 32
3.2 Overview of embedded OS . . . 32
3.2.1 OS middleware in embedded systems . . . 32
3.2.2 Embedded OS services and functionalities . . . 33
3.3 Experimental setup . . . 38
3.3.1 OMAP3530 Applications Processor . . . 39
3.3.2 OMAP3530 EVM board . . . 39
3.3.3 Measurement framework . . . 39
3.4 Energy characterization and estimation flow . . . 41
3.5 OS power and energy modeling . . . 44
3.5.1 Scheduling routines . . . 45
3.5.2 Context switch . . . 46
3.5.3 Inter-process communication . . . 53
3.6 Conclusion . . . 56

In this chapter, a flow of embedded OS power/energy consumption characterization is introduced. First, an overview of embedded operating system is presented. Then, we detail the methods used to determine energy and power overheads of three basic services of the embedded OS: scheduling, context switch and inter-process communication. Also, the variation of power/energy consumption of these embedded OS services is studied. Furthermore, the impact of hardware and software parameters like processor frequency and scheduling policy on energy consumption is analyzed. Mathematical models for power and energy consumption are extracted. The use-case embedded system used is the OMAP3530EVM board with an OMAP3 processor and Linux 2.6.32 operating system.

3.1 Introduction

As mentioned previously in last chapter, embedded systems become so complex as they contain various hardware devices and software application which interact with users to handle these systems. The complexity of hardware and software layers necessitates the use of a specific support that allows the application to exploit efficiently the hardware platform. This support is the embedded OS; it includes libraries and device drivers and offers a wide variety of services. Estimating and modeling the energy consumption of OS routines and services constitute a challenge for embedded system designers. In this chapter, First, we propose a flow of OS energy characterization. We study the variation of the energy and power consumption of the embedded OS services. We detail the methods used to determine power/energy overheads of embedded OS basic services: scheduling, context switch and inter-process communication. Then, we analyze the impact of hardware and software parameters like processor frequency and scheduling policy on the OS energy consumption in order to deduce models and laws that estimate this consumption.

3.2 Overview of embedded OS

In this section, an overview of embedded OS is presented. We present the OS middleware in embedded systems and detail its different services.

3.2.1 OS middleware in embedded systems

In embedded systems, the OS serves as an interface between the software application and the hardware platform. It is an important software component in many embedded system applications since it drives the exploitation of the hardware platform by offering a wide variety of services: task management, scheduling, inter-process communication, timer services, I/O operations and memory management. Also, the embedded OS manages the overall power consumption of embedded system components. It includes many power management policies aiming at keeping components into lower power states, thereby reducing energy consumption.
Figure 3.1 shows the disposition of embedded systems' different layers. The application is represented by a set of (n) tasks $\{Task\ \mathrm{i}, 1 \leq i \leq n\}$. The embedded OS includes (m) services used by the application to exploit the hardware platform resources. The set of these services is $\{S\ \mathrm{j}, 1 \leq j \leq m\}$.
In next section, main functionalities and services of embedded OS are detailed.

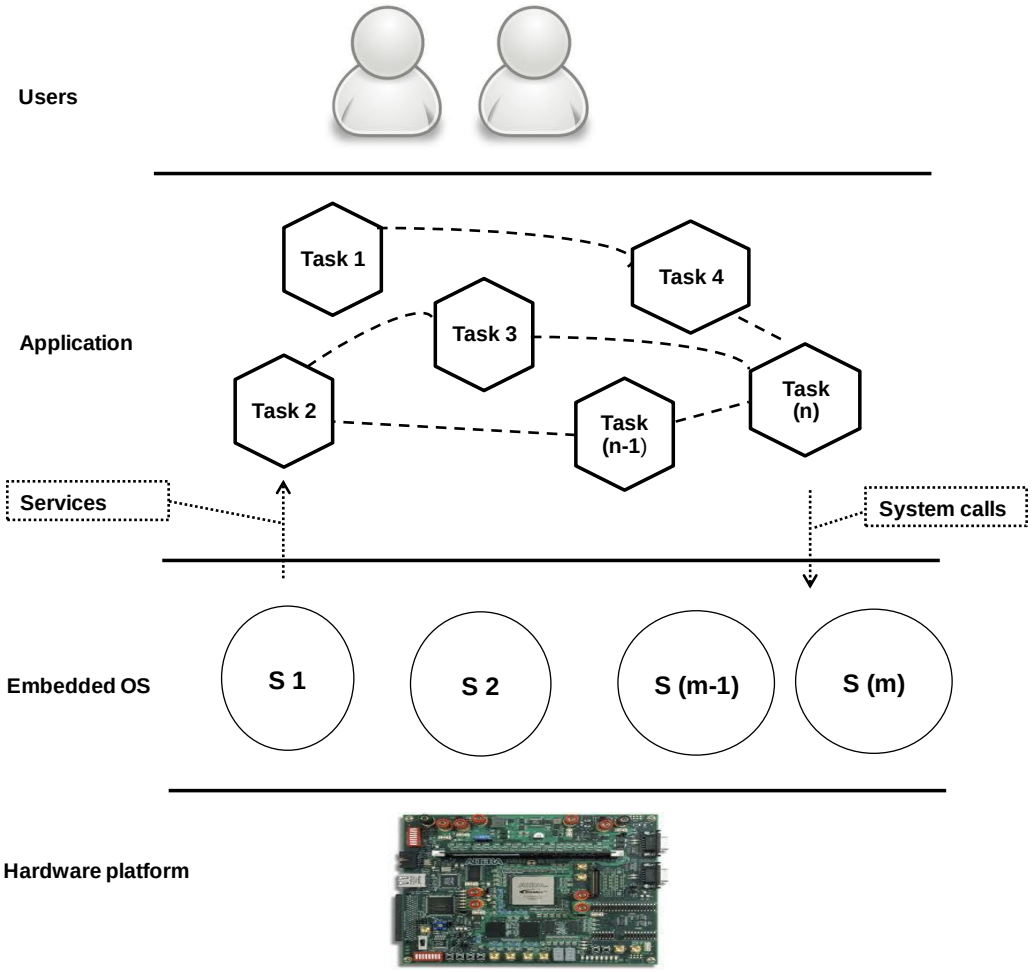


Figure 3.1: The different layers of an embedded system

3.2.2 Embedded OS services and functionalities

To bind the software application tasks on the hardware platform components, the embedded operating system provides various services and functionalities. The main services are detailed below.

- **Interprocess communication and synchronization:**
  This service is called when two or more processes need to communicate with each other. The OS ensures the data exchange, resources share, and synchronization between these processes. The process synchronization access is achieved using signals, mutexes and semaphores. The OS uses different technique for data exchange between the processes, such as, named pipes, anonymous pipes, message queues and shared memory. These mechanisms will be

detailed later when characterizing the energy consumption of this service.

- **Clock/timer functions:** Embedded OS uses heavily the timer functions to schedule the different processes. This service could be exploited by the user till it provides a set of basic functions such as getting the current time and the elapsed time.

- **Device management:** This service consists in handling peripheral devices through the processor using a set of commands and signals. The component that makes these commands easily understandable by hardware devices is named the device controller. The latter is an interface between the OS and the peripheral device. The OS software routines that control each device is called device driver. The OS needs many device drivers to ensure the proper functioning of different peripheral devices. For this reason, when a new peripheral is added to the embedded system, its device driver should be integrated in OS code.

- **Memory management:** The OS includes a unit called memory management unit that manages the accesses to embedded system memory requested by the processor. This unit allocates dynamically the memory when the application tasks need it. Then, it frees the memory when they are not required for reuse. Also, the OS handles the swapping between main memory and disk when the main memory is too small to hold all the data that needs the application for execution.
  Besides, as showed in figure 3.2, the memory management unit uses the virtual memory mechanism to convert the logic or virtual addresses, which are used by the processor, to a physical addresses that allow the access to various memory locations called pages. This address conversion is performed via an associative cache called translation look-aside buffer (TLB).

- **Tasks handling:** This service controls the different states of applicative tasks. These tasks are represented by processes or threads. Processes have distinct address spaces, while threads share the same address space inside a process. The OS handles the creation, the execution and the termination of application tasks. The task has three possible states: running, blocked and ready states. When the task executes its routines on the processor, it is considered in running state. A task is in ready state means that it is ready for execution but can not run because the processor is used by another task. An application task is in blocked state when it is unable to run until some external event happens; for instance, it is waiting for a resource to be available. Different transitions between task's states are showed in figure 3.3.

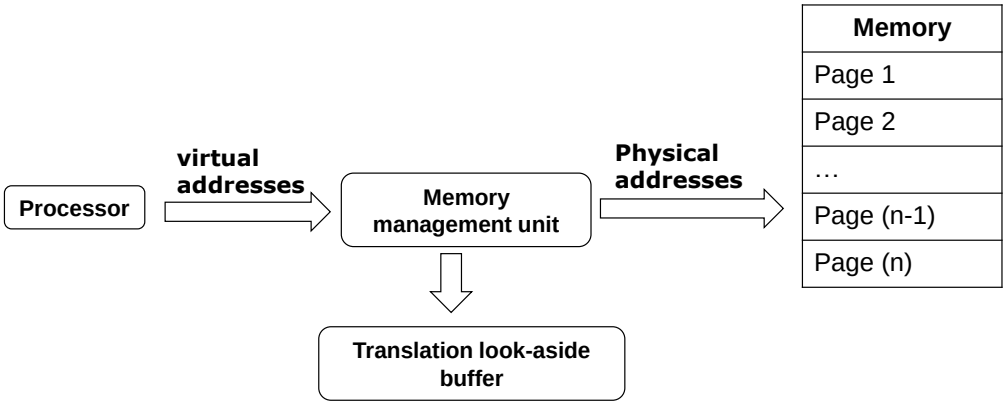


Figure 3.2: Address conversion by memory management unit

- **Tasks scheduling:** The OS makes the application tasks scheduling decision. It includes a module that defines which tasks are running on the processing platform at every time instant. Scheduling routines determine the interleaving of execution for application tasks on the target processor. This interleaving is named a schedule. The schedule must be produced to ensure that every job of task executes on processor(s) for its execution requirement (WCET) during its scheduling window.
  A scheduling event is occurred generally at various situations. In fact, the scheduler is called when a process finishes its execution so that it can no longer run on the processor. As a result, the OS selects another process from the list of ready processes. When there is no a ready process, an idle process is chosen for running. Moreover, when a process blocks on a resource, the scheduler chooses another process for execution. Also, the scheduler is called, when a new process is created, to run the parent or the child process. Furthermore, the OS invokes the scheduler when timer provides periodic interrupts. A scheduling decision can be taken after each timer interrupt or after every n timer interrupts.
  The choice of the next process to run depends on the scheduling algorithm. The scheduling policies aim at increasing the processor efficiency by maximizing the number of tasks completed per unit of time and reducing the average waiting time of different tasks. Scheduling algorithms can be classified based on two main criteria: the execution and preemptive-ness of tasks [73][96].
  Consequently, scheduling algorithms are also divided into two categories based on the tasks execution criteria: the off-line and online scheduling algorithms. The off-line scheduling algorithms assign different tasks to processor before the execution step. These algorithms are usually carried out via a scheduling

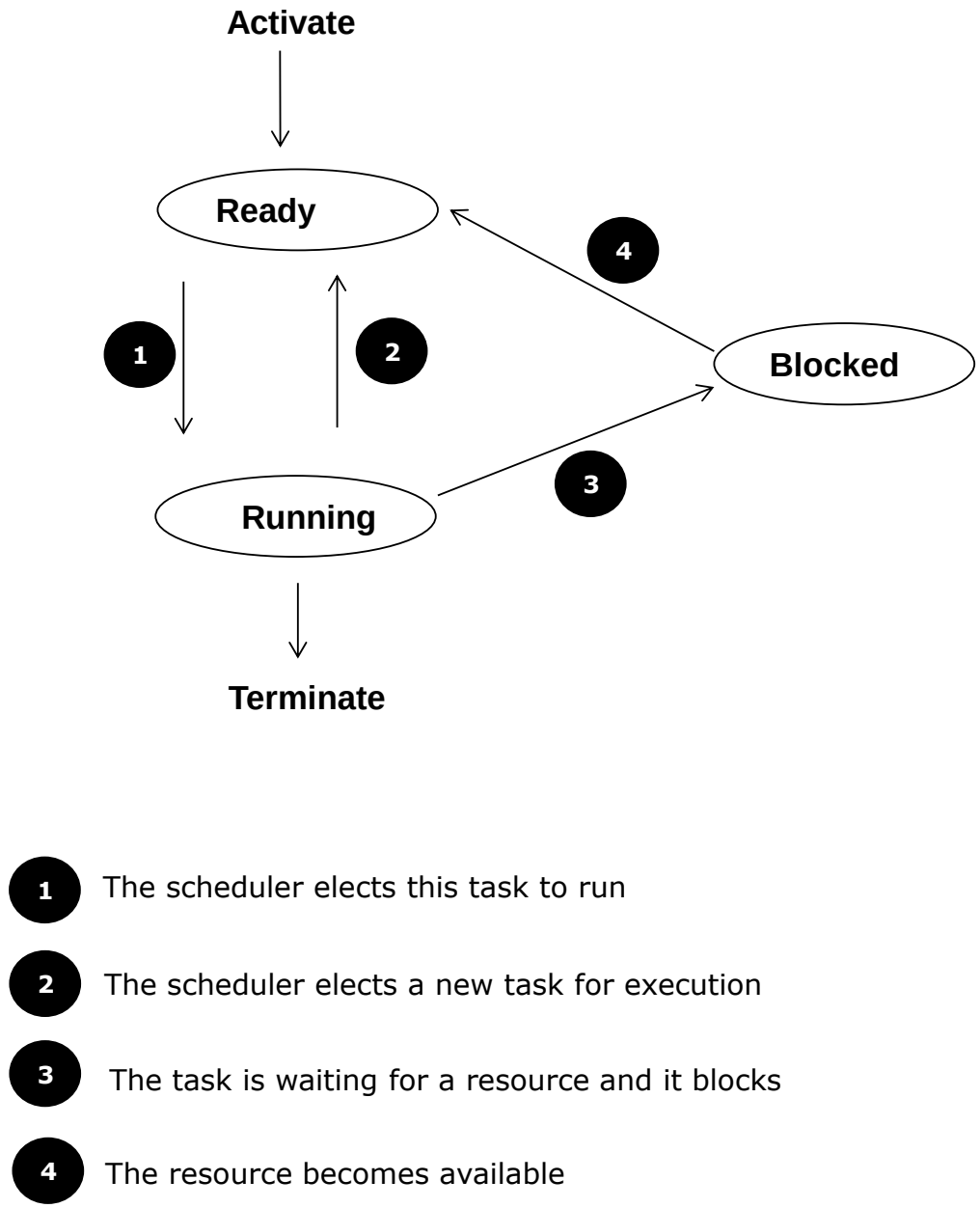


Figure 3.3: Transitions between task states

table describing the predetermined schedule. This table contains the set of tasks and their activation times. The off-line scheduler is called every time period. For example, if there is a set of periodic tasks to be scheduled, an off-line schedule may be generated for an interval of length equal to the least

common multiple of the periods of different tasks. Unlike off-line schedulers, online scheduling algorithms examine the active tasks properties and make the scheduling decision when running tasks.
Furthermore, based on preemptive-ness of tasks criteria, scheduling algorithms can be broadly classified into non-preemptive and preemptive algorithms. A non-preemptive scheduling algorithm elects a task for execution and do not interrupt its execution until it blocks or until it voluntary releases the CPU. But, a preemptive scheduling algorithm uses an interrupt technique to suspend the currently executing process and elects a new process for execution. Therefore, all processes will get some amount of CPU time at any given time. The main scheduling policies in classical operating systems are: First-in-First-out (FIFO), round-robin. These policies will be detailed later when characterizing the energy consumption of scheduling routines. Also, in order to respect the deadlines and time constraints, various scheduling policies are used in real-time OS such as the rate monotonic (RM) and earliest deadline first (EDF) policies [42]. Also, to reduce the energy consumption, low power scheduling policies are used, such as (DPM) and (DVFS) policies. These policies will be detailed later in chapter 5.

- **Context switch:** The context switch is a mechanism which occurs when the kernel changes the control of the processor from an executing process to another that is ready to run. The kernel saves the state of current process including the processor register values and other data that describe this state. Then, it loads the saved state of the new process for execution. This service will be detailed later when estimating its energy consumption.

- **Error detection:** The OS is able to detect the different errors that could occur when the embedded system is running. The source of the errors could be the processor, the memory, the peripheral devices and the user programs. When an error occurs, the OS informs the user about the malfunctioning of the system and indicates the cause of the error. Then, the OS takes the convenable action to correct this error.

- **User interface:** To handle easily the hardware platform, this service provides an interface between the user and the hardware components. This interface could be a command line interface through a command line interpreter, such as text terminal, or a graphical user interface using graphical icons and elements.

Figure 3.4 synthesizes the interactions between the OS services and the hardware components.

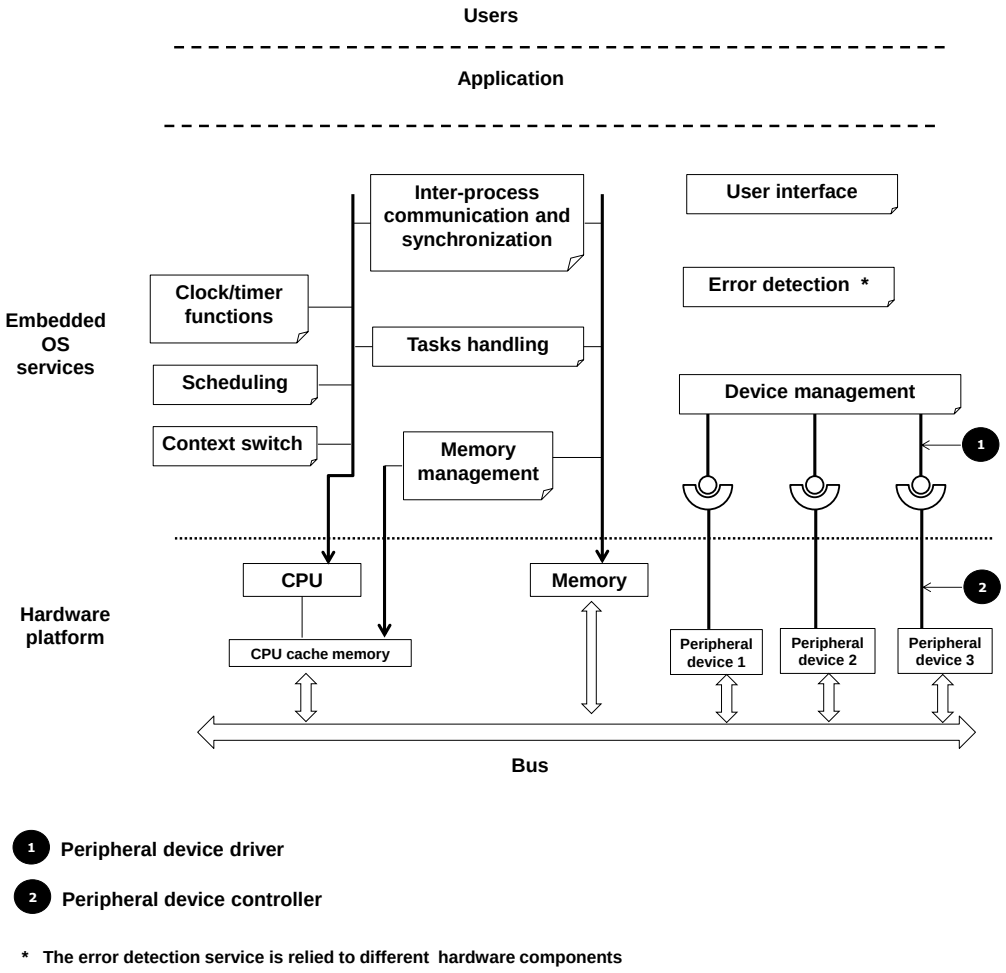


Figure 3.4: The exploitation of hardware components by the OS services

3.3 Experimental setup

Under the OPEN-PEOPLE project, academic and industrial partners have chosen the OMAP35x EVM board [7] as a hardware platform to validate this work. The proposed approach is generic and could be applied to other hardware platforms and OSs.

The OMAP (Open Multimedia Applications Platform) architecture, developed by Texas Instruments [14], is a category of proprietary system on chips for different multimedia applications. The OMAP processors are used by various devices such as Samsung, Nokia, and Motorola mobiles [63].

To characterize the energy overhead of embedded OS, we use the OMAP35x Evaluation Module (EVM) board, equipped with OMAP3530 processor, as an embedded system. This board builds low power applications requiring low power consumption

and high performance such as portable media players, navigation devices, software defined radio, medical applications and media controllers.
In this section, we describe and detail the characteristics of hardware and software components of this board.

3.3.1 OMAP3530 Applications Processor

The OMAP3530 multimedia applications processor is developed based on advanced Super-scalar 720 MHz ARM Cortex-A8 RISC Core and a digital signal processor (520 MHz TMS320C64x DSP).
Figure 3.5 shows the different components of OMAP3530 processor. These are characteristics of this processor:
- CPU: ARM Cortex-A8 RISC
- Operating frequencies: 125, 250, 500, 550 and 720 MHz
- External Memory Type Supported: LPDDR, NOR Flash, NAND flash, One-NAND, Asynch SRAM DMA64-Ch E
- Core Supply: 0.8 V to 1.35 V
- IO Supply:1.8 V,3.0 V
- IVA2.2 subsystem with a C64x+ digital signal processor (DSP) core
- POWER SGX subsystem for 3D graphics acceleration to support display and gaming effects

3.3.2 OMAP3530 EVM board

The different components of the OMAP35x EVM board are the processing subsystem (including one or more processor cores, hardware accelerators, etc), a memory subsystem, peripherals as well as global and local interconnect structures (buses, bridges, etc). Figure 3.6 and figure 3.7 show the different features of the OMAP35x EVM Board. The embedded OS used is *linux-omap* which is supported for use with the OMAP35x EVM.
In this work, we are interested in studying the energy overhead of the processor core supporting the OS, the ARM Cortex-A8 processor.

3.3.3 Measurement framework

When power is first applied to OMAP35x EVM board, many hardware elements are initialized before the execution and running of user application and OS routines. This early initialization code is a part of the boot-loader. After the hardware platform initialization step, the boot of OS image is performed using the boot-loader. Then, Once the OS has started execution, it takes control of the board and the boot-loader is overwritten and ceases to exist.

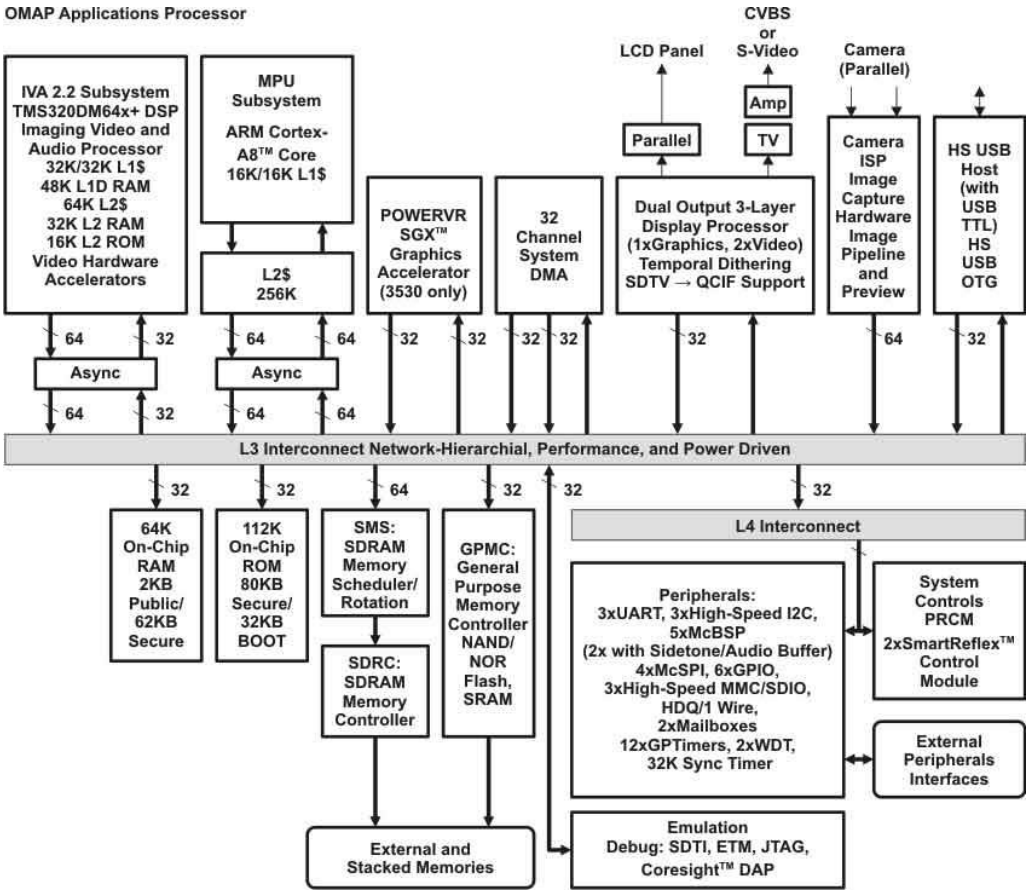


Figure 3.5: OMAP3530 Applications Processor [7]

The "U-Boot" open-source boot-loader for the OMAP35x EVM board is generated. It includes support for ethernet interfaces and supports several network protocols such as BOOTP, DHCP, TFTP and NFS. The "U-Boot" is also able to update the board embedded flash memory with an image downloaded through the ethernet.
As shown in figure 3.8, the measurement platform includes a dedicated server to configure the OMAP35x EVM board and to control the energy consumption measurements on this board. It consists of a computer wire-connected to the board. We use the DHCP protocol to obtain an IP address of the board from the server. The TFTP (Trivial File Transfer Protocol) and NFS (Network File System) protocols are used to load and boot the OS image from the server and through the ethernet. The test programs are executed on the hardware platform and the energy dissipated by the processor is determined as follows: the voltage drop V_{drop} across a jumper J6 pins connected in series with the OMAP 3530 processor is measured. Then, the current consumed is calculated after dividing V_{drop} by a shunt resistance R in

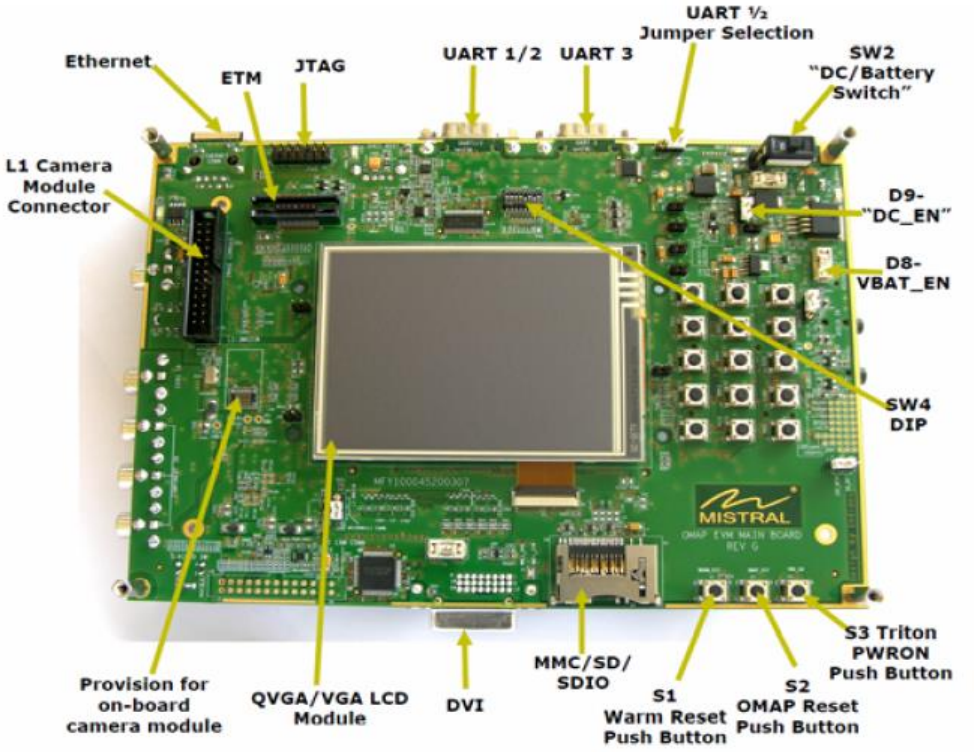


Figure 3.6: OMAP35x EVM Board top overview

parallel with the jumper pins.

3.4 Energy characterization and estimation flow

The proposed method targets to extract models of embedded OS services power/energy overhead. The inputs are the embedded OS, the application and the hardware platform. As showed in figure 3.9, to characterize the energy consumed by OS services, a set of benchmarks, which are test programs that stimulate each service separately, are implemented. These programs are compiled and linked to the OS.

In the energy analysis step, a set of parameters are varied: hardware and software parameters which influence the energy consumption are identified then energy profiles are traced. The energy traces obtained are able to characterize the energy overhead of the OS services and then to model the power and energy consumption. After extracting energy models, we estimate the energy and power overheads, as showed in figure 3.10. We focus on the correlation between the energy consumed

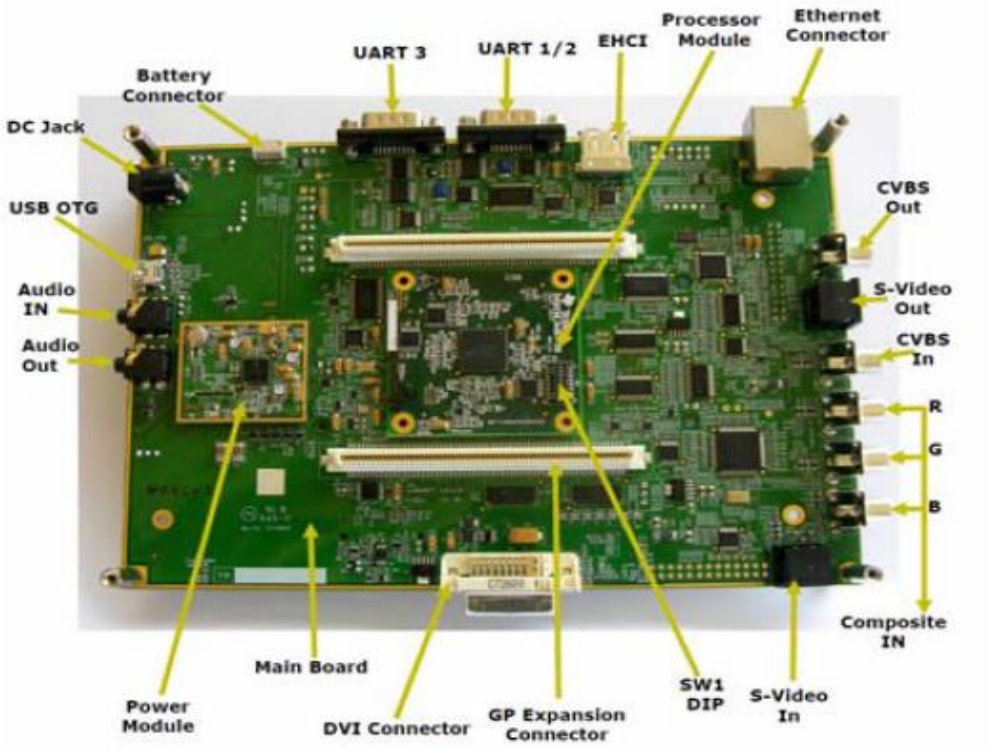


Figure 3.7: OMAP35x EVM Board bottom overview

by the application and the OS services. The energy consumption of an applicative task is depicted by equation 3.1.

$$\forall T_i, E_{T_i} = Eintra_i + (\sum_{1 \leq j \leq p} \delta_{i,j} \times E_{S_j}) \tag{3.1}$$

Where E_{T_i} represents the energy consumed by the task T_i, $Eintra_i$ is the energy consumed by this task routines and operations, p is the number of services used by the task T_i, $\delta_{i,j}$ is energy consumption rate of the task T_i using the service S_j and E_{S_j}, the energy consumption of the service S_j.
We consider t the total number of the OS services, x_j the number of the parameters that influence E_{S_j}, the energy consumption of the service $S_j, 1 \leq j \leq t$.
The set of parameters appropriate to the energy overhead of the service S_j is $\{ Param_{j,k}\ , 1 \leq j \leq t, 1 \leq k \leq x_j \}$. The function f_k describes the variation of E_{S_j} with $Param_{j,k}$. We compute the energy consumption of the service S_j following

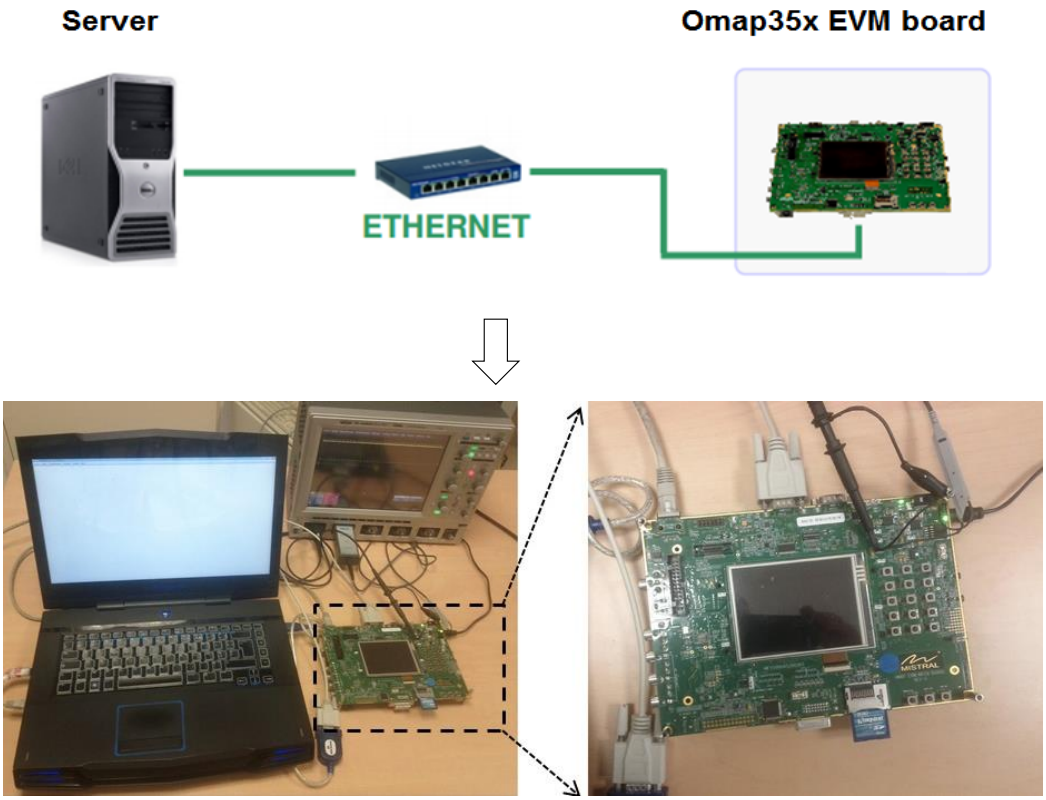


Figure 3.8: The measurement framework

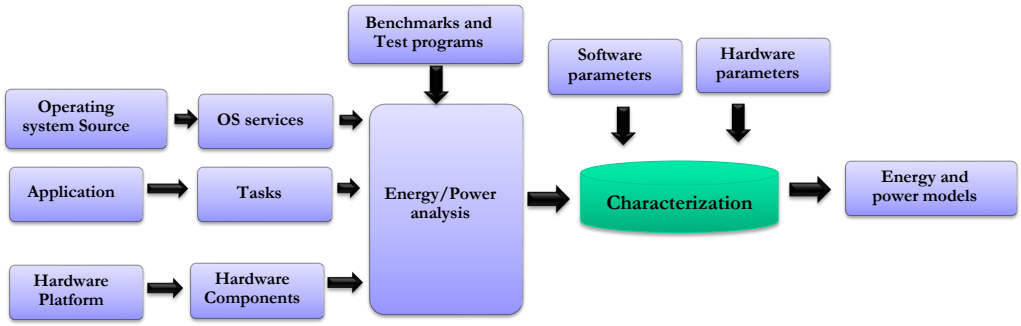


Figure 3.9: The methodology of OS energy characterization

this equation 3.2.

$$E(S_j) = \sum_{1 \leq k \leq x_j} f_k(Param_{j,k}) \tag{3.2}$$

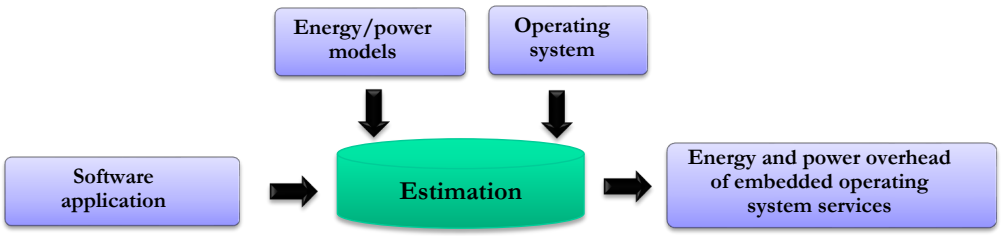


Figure 3.10: Estimation of the operating system energy consumption

According to the amount of energy consumed by each service when executing the application, we classify the operating system's energy overhead into two groups: a basic energy consumed by basic OS services. Each basic service consumes an amount of energy bigger than an energy threshold E_{th}, this threshold is the average energy consumed by the different OS services. The remaining services are the secondary services. The set of basic and secondary services are respectively $\{BS_j, 1 \leq j \leq p\}$ and $\{SS_k, 1 \leq k \leq q\}$ where p and q are respectively the number of basic and secondary services.
Expression 3.3 depicts the energy consumed by the operating system E_{OS} when running a task T_i. Equation 3.4 verifies whether the totality of the energy consumed by the OS when running the application, having n tasks, is well distributed between different services.

$$E_{OS} = (\sum_{1 \leq j \leq p} \alpha_{i,j} \times E_{BS_j}) + (\sum_{1 \leq k \leq q} \beta_{i,k} \times E_{SS_k}) \tag{3.3}$$

Where

$$\sum_{1 \leq i \leq n} (\sum_{1 \leq j \leq p} \alpha_{i,j} + \sum_{1 \leq k \leq q} \beta_{i,k}) = 100\% \tag{3.4}$$

$\alpha_{i,j}$: energy consumption rate of the task T_i using the service BS_j.
$\beta_{i,k}$: energy consumption rate of the task T_i using the service SS_k.
E_{BS_j} and E_{SS_k} represent respectively the energy consumed by the service BS_j and SS_k.
In next section, we will detail methodologies and benchmarks used to characterize the embedded OS services energy overhead and study its variation with hardware and software parameters.

3.5 OS power and energy modeling

In this section, embedded OS services energy characterization approaches are introduced, three important services are studied: the scheduling, the context switch and

inter-process communication.

3.5.1 Scheduling routines

Scheduling routines and operations could generate power overhead on the processor and/or memory components. They are considered as system calls and only consist in switching the processor from unprivileged user mode to a privileged kernel mode. To quantify power and energy overhead of embedded OS scheduler routines and operations, we have to build test programs containing threads with different priorities, we measure in a first step the average energy consumed by the stand-alone tasks without scheduling routines, and then with scheduling routines.
$E_{Scheduling}$ represents the energy consumed by the scheduling operations. It is calculated as showed in equation 3.5:

$$E_{Scheduling} = E_{withsch} - E_{withoutsch} \tag{3.5}$$

Where $E_{withsch}$ and $E_{withoutsch}$ represent respectively the energy consumed by the benchmarks with scheduling routines and without scheduling routines.
We vary several parameters when running the test programs. The applicative parameter that we can change is the scheduling policy. We also modify the processor frequency as a hardware parameter. We are interested in studying the influence of three scheduling policies: $SCHED_FIFO$, $SCHED_RR$ and $SCHED_OTHER$. The used embedded OS supports only these scheduling policies.
$SCHED_FIFO$ policy is used with static priorities higher than 0, it is a scheduling algorithm without time slicing. Under this policy, a process which is preempted by another one having higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. If there are two $SCHED_FIFO$ processes having the same priority, the process which is running will continue its execution until it decides to give the processor up. The process having the highest priority will use the processor as long as it needs it.
$SCHED_RR$ policy enhances the $SCHED_FIFO$ one; hence, everything described above for $SCHED_FIFO$ also applies to $SCHED_RR$ except that each process is only allowed to run for a maximum time called quantum. If a $SCHED_RR$ process has been running for a time period equal to or greater than the time quantum, it will be put at the end of the priority list. Only fixed-priority threads can have a $SCHED_RR$ scheduling policy. A $SCHED_RR$ process that has been preempted by a higher priority process subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum.
$SCHED_OTHER$ policy is only used at static priority 0. To ensure a fair progress among the processes, the $SCHED_OTHER$ scheduler elects a process to run from

the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level and increased for each time quantum, when the process is ready to run, but denied to run by the scheduler.
Figure 3.11 shows the evolution of the power overhead of the scheduler routines $P_{Scheduling}$ over the scheduling policy. We can see that the energy consumed when we use $SCHED_OTHER$ policy is important compared to $SCHED_FIFO$ and $SCHED_RR$ policies. This is due to the additional operations (*nice* or *setpriority*() system calls) used when the $SCHED_OTHER$ scheduler calculates and increases the dynamic priority for each time quantum. $P_{Scheduling}$ increases with the rise of the number of processes, this is due to the increase of the scheduler routines, such as the assignment of the priorities.
Figure 3.12 depicts the variation of measured and estimated scheduling routines energy consumption with processor frequency. The used frequencies are operating points of the processor: 125 Mhz, 250 Mhz, 500 Mhz and 720 Mhz. The scheduling policy is $SCHED_OTHER$ and the number of processes is 10. The energy consumption law for the scheduling routines is depicted by equation 3.6. The average estimation error is around 0.355%.
The obtained results are explained by considering that the energy is the product between the average power and the total execution time. If we consider that the steady state current (and hence the power) profile obtained when running this experiment is almost flat since the processor does not access the external bus, the energy cost of the scheduler is proportional to the execution time of the scheduling routines which decrease with the increase of the frequency.

$$E_{scheduling}(f) = (-59.649 \times 10^{-3} \times f) + (3.106 \times 10^{2}) \tag{3.6}$$

3.5.2 Context switch

Context switching is a fundamental mechanism and one of basic services of the embedded OS, it ensures the share of processor resources across multiple threads of execution. This mechanism consists in saving the processor state of a thread and loading the saved state of another thread.
In the majority of recent works presented in last chapter, the authors do not take into account the energy and time overheads of this service when studying the energy consumption of the operating systems. They include it with the scheduling service, but the two services are distinct. Actually, in embedded systems, the processor has two operating modes: the kernel mode and user mode. The processes running on kernel and user mode are called respectively kernel and user processes. The user process runs in a memory space which can be swapped out when necessary. When the processor needs the user process to execute a kernel code, the process becomes

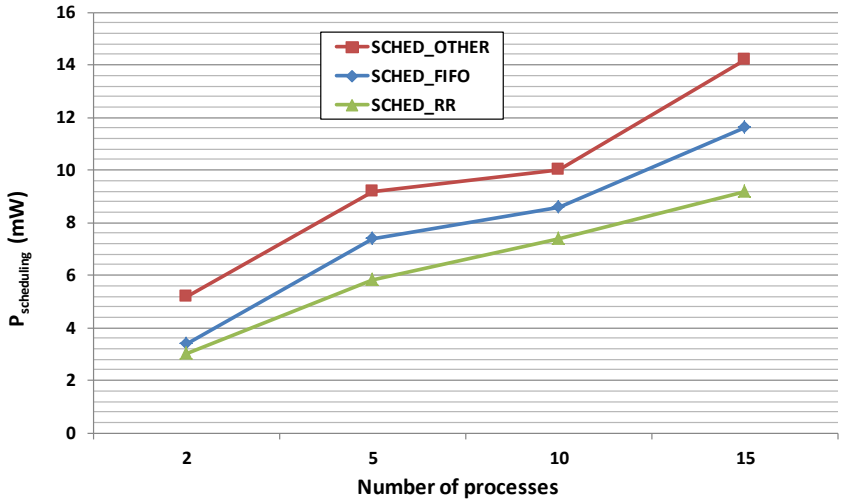


Figure 3.11: Scheduling routines power consumption versus the number of processes for different scheduling policies

in kernel mode with administrative privileges. In this case, the processor has no restrictions while executing the instructions and will access to key system resources. Once the kernel process finishes its workload, it returns to the initial state as a user process. The scheduler switches the processor from the user mode to a kernel mode via system calls; this mechanism is named the mode switch. Unlike the mode switch, the context switch consists in switching the processor from one process to another.

The context switch service introduces direct and indirect overheads [68]. Direct context switch overheads include saving and restoring processor registers, flushing the processor pipeline and executing the OS scheduler. Indirect overheads involve the switch of the address translation maps used by the processor when threads have different virtual address spaces. This switch perturbs the TLB (CPU cache that memory management hardware unit uses to improve virtual address translation speed) states. Also, the indirect context switch includes the perturbation of the processor's caches. In fact when a thread T_1 is switched out and a new thread T_2 starts the execution, the cache state of T_1 is perturbed and some cache blocks are replaced. So, when T_1 resumes the execution and restores the cache state, it gets a cache misses. Besides, the OS memory paging represents a source of the indirect overhead since the context switch can occur in a memory page moved to the disk when there is no free memory.

Prior research has shown that indirect context switch overheads [110], mainly the

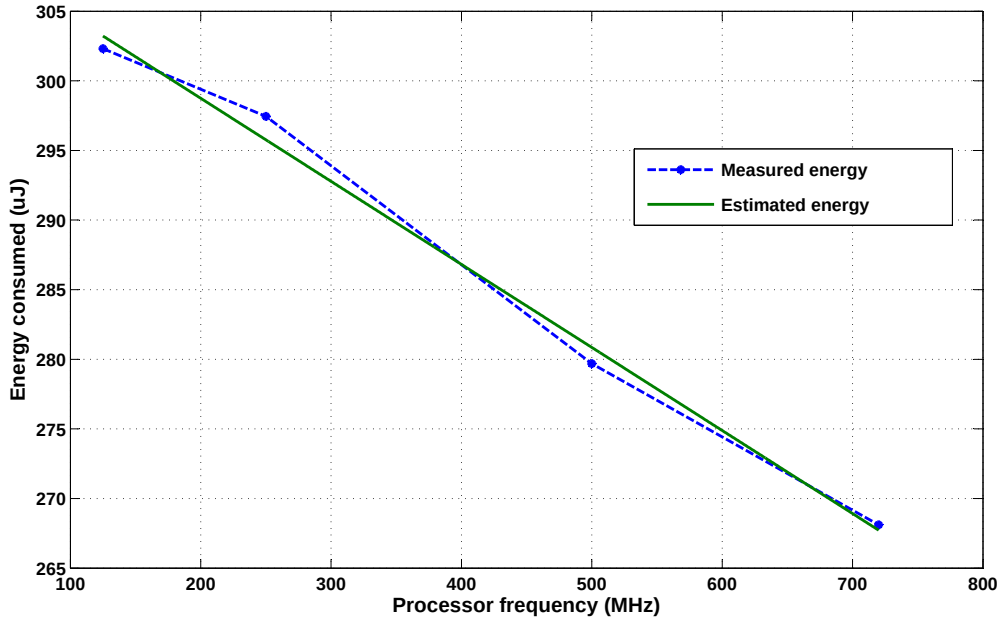


Figure 3.12: Scheduling routines energy variation as a function of CPU frequency ($SCHED_OTHER$ policy and 10 processes)

cache perturbation effect, are significantly larger than direct overheads.
To characterize the energy consumption of the context switch, we create a set of threads in a multitasking environment using the POSIX standard [113].
As depicted in figure 3.13 and figure 3.14, the test-bench consists in creating two threads P1 and P2 and generating a number of context-switches as detailed in our recent work [79]. In fact, in step 1, only one context switch is generated and in step n, n context switches are generated.
In the remainder of this dissertation, Tcs represents the time of the context switch, $S_{i,j}$ the j-th section of the process P_i and $T_{i,j}$ is the execution time of the section $S_{i,j}$.

The total execution time of the benchmark in step 1 and step n are respectively T_{step1} and T_{stepn}. They are depicted by equations 3.7 and 3.8:

$$TStep1 = Texec_1 + Tcs + Texec_2 \tag{3.7}$$

$$TStepn = \sum_{1 \leq i \leq p} T_{1,i} + \sum_{1 \leq j \leq q} T_{2,j} + (n \times Tcs) \tag{3.8}$$

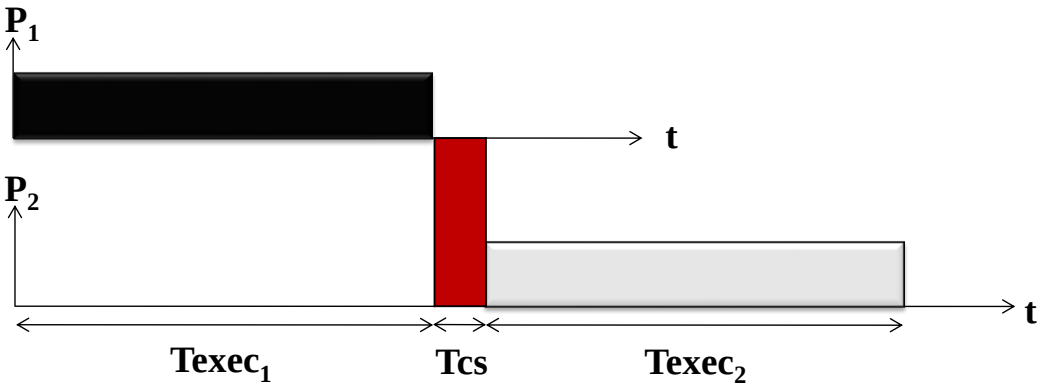


Figure 3.13: Step 1

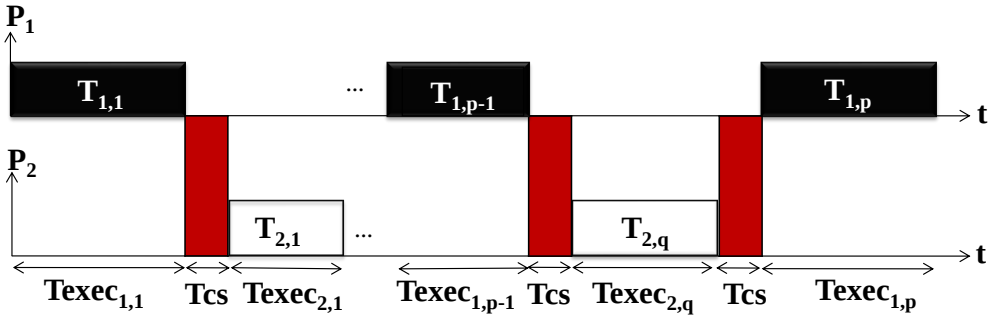


Figure 3.14: Step n

Where p and q represent respectively the number of sections of P_1 and P_2.
The context switch time Tcs and the context switch power overhead Pcs are calculated following equations 3.9 and 3.10:

$$Tcs = (T_{stepn} - T_{step1})/(n-1) \tag{3.9}$$

$$Pcs = (P_{stepn} - P_{step1})/(n-1) \tag{3.10}$$

The context switch energy overhead is computed as showed in equation (3.11):

$$Ecs = ((P_{stepn} * T_{stepn}) - (P_{step1} * T_{step1}))/(n-1) \tag{3.11}$$

Where P_{step1} and P_{stepn} are respectively the average power consumption of the benchmarks in step 1 and step n.
We execute the test programs following the characterization approach. Then, we

vary the scheduling policy and the frequency, we note the power and performance variations and we extract energy models.

3.5.2.1 The scheduling policy impact on the context switch overhead

In our experiments, the scheduling policy and the number of context switches are varied and the energy consumed is measured when a context switch occurred. The variation of the energy dissipated according to the number of context switches and the scheduling policy is presented in figure 3.15. This figure compares the decrease of the context switch energy overhead for the two processes, P1 and P2, by varying the number of context switches.
It is noted that the context switch energy overhead decreases with the increase of the number of context switches. In fact, when the first context switch from one process to another occurs, a data structure named Process Control Block (PCB) is created in order to save the state of each process. The energy overhead of the creation of the PCB is accounted with the context switch energy overhead and is divided between the context switches. As a result, if the number of context switches increases, the average Ecs per context switching decreases. Also, when the scheduling policy used is $SCHED_FIFO$, the context switch energy overhead is more important than the energy for the $SCHED_RR$ scheduling policy. Actually, under the round robin scheduling policy, the processor assigns time slices (quantum) to each process. So, before the context switches that we generate, there is another context switches that occurred automatically due to the expiration of the quantum of the process P1. Consequently, the PCB is created during the automatic context switch. The energy overhead of the PCB creation is not accounted with the energy of the context switch that we generate: Ecs. But, under the FIFO scheduling policy, the processor does not switch automatically from the process P1 to P2 only if P1 terminates its execution so that the energy overhead of the PCB creation is accounted with Ecs.
We note that $SCHED_OTHER$ processes are non real time processes, but, $SCHED_RR$ and $SCHED_FIFO$ are real time processes. So, $SCHED_RR$ and $SCHED_FIFO$ processes need more memory than $SCHED_OTHER$ processes to save the processor registers because they perform more operations and calculations in order to respect the real time constraints. Consequently, they consume more time to change the context. Then, the context switch of the $SCHED_OTHER$ processes consume less energy than the $SCHED_RR$ and $SCHED_FIFO$ ones.

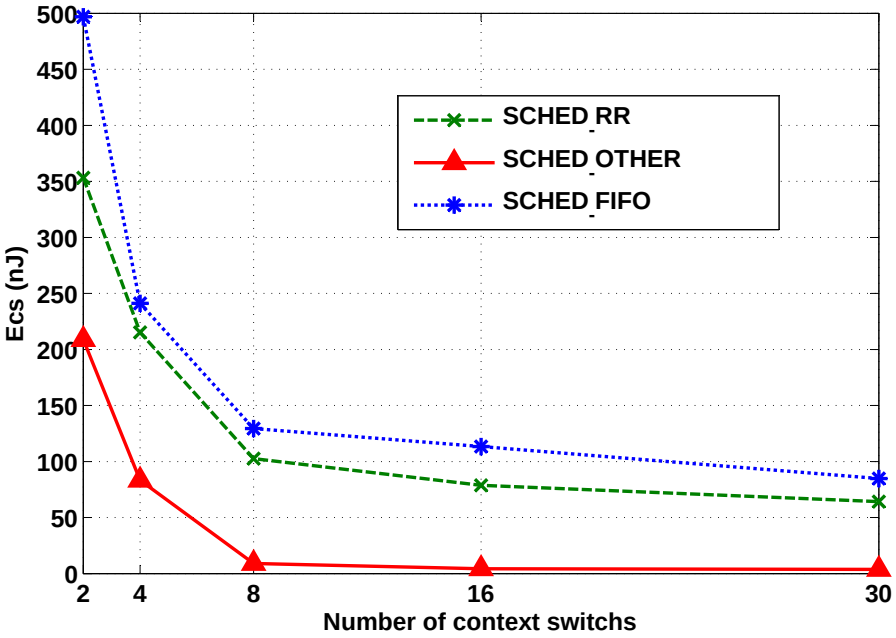


Figure 3.15: Context switch energy consumption versus the number of context switching for different scheduling policies

3.5.2.2 The processor frequency impact on the context switch overhead

The CPU frequency is another parameter that could influence significantly the energy consumption of the context switch. In this paragraph, the impact of processor frequency on the context switch energy overhead for static and dynamic frequency cases is discussed.

- **Static frequency case:** For this experiment, the scheduling policy and the number of context switches are fixed. The benchmarks of step 1 and step n with a static frequency are executed. The CPU frequency is varied afterwards and the benchmarks are re-executed.
  As explained in chapter 2, in CMOS technology-based systems, there are two principle sources of power dissipation: dynamic power dissipation, which arises from the repeated capacitance charge and discharge on the output of the hundreds of millions of gates in modern chips, it depends on the processor frequency, and static power dissipation which arises from the electric current that leaks through transistors even when they are turned off. The hardware platform used in the current work reduces standby power consumption by reducing power leakage so that static power is negligible compared to the dy-

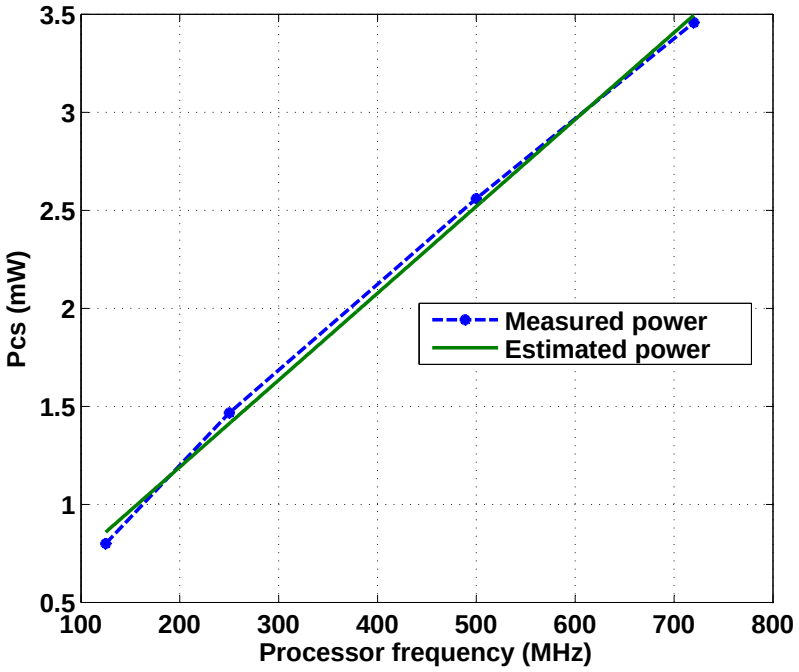


Figure 3.16: Context switch power variation as a function of CPU frequency

namic power. Figure 3.16 plots the measured and estimated context switch power, mainly the dynamic power, consumption as a function of the frequency. Context switch power variation with processor frequency follows the law presented in equation 3.12. The average error of the proposed methodology results against the physical measurements is about 3.4%.

$$Pcs(f) = (4.4 \times 10^{-3} \times f) + 0.3041 \tag{3.12}$$

Where f is the CPU frequency, the unit of Pcs and f is respectively mW and MHz.

The voltage V_{drop} across the processor increases with the rise of the processor frequency so that the power consumption increases with the frequency.

- **Dynamic frequency case:** The core frequency is dynamically changed during the execution of benchmarks: test programs are executed in step 1 and step n. The processes P1 and P2 are executed respectively at a frequency F1 and F2. When the processor preempts the process P1 and executes the process P2, the core frequency changes from F1 to F2; and inversely.

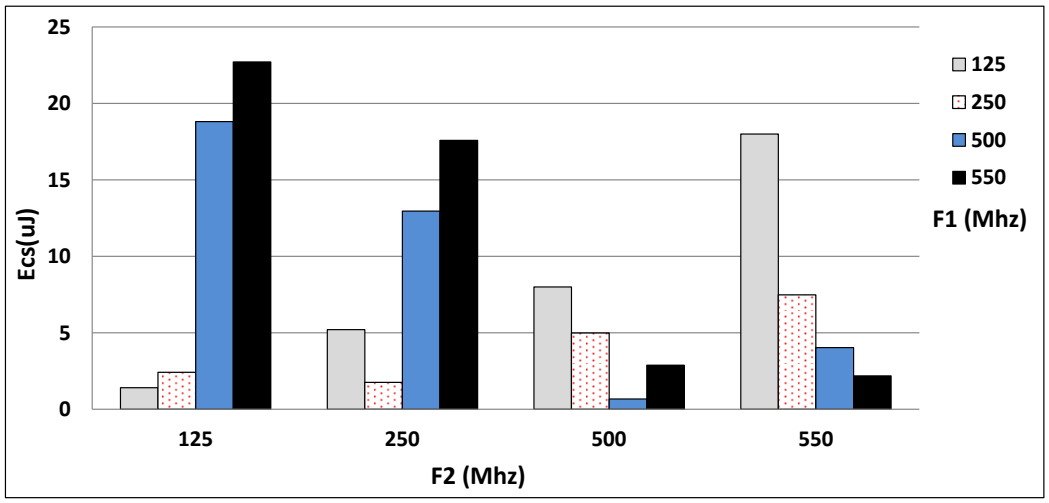


Figure 3.17: Context switch energy variation as a function of dynamic CPU frequency scaling

Figures 3.17, 3.18 and 3.19 illustrate the context switch energy, power and time variation when changing the CPU frequency. Actually, for the processor core, a set of voltage and frequency couples is specified, named operating points. Running on high frequency requires also high voltage and inversely. For raising the frequency and supply voltage, the microprocessor sets a new VID (voltage identifier) code to have a higher output voltage than the current one, and conversely. This operation leads to time and energy overhead [80]. Also, the more important the difference between F1 and F2 is, the higher context switch energy is. This is due to the perturbation of the processor's cache memory resulting from the variation of processor bus frequency which varies with the processor frequency.

3.5.3 Inter-process communication

Inter-process communications (IPC) allow threads in one process to share information with threads in other processes, and even with processes that exist on different hardware platforms. The embedded OS explicitly copies information from sending process's address space into a distinct receiving process's address space. Examples of IPC mechanisms are pipes, message passing through mailboxes and shared memory. To characterize the power and energy consumption of IPC, we have to execute test programs, each one repeatedly calling an IPC mechanism. The aim is to get an average power and performance overhead of the IPC mechanism when running the programs. The set of parameters that we could vary are: the type of IPC mecha-

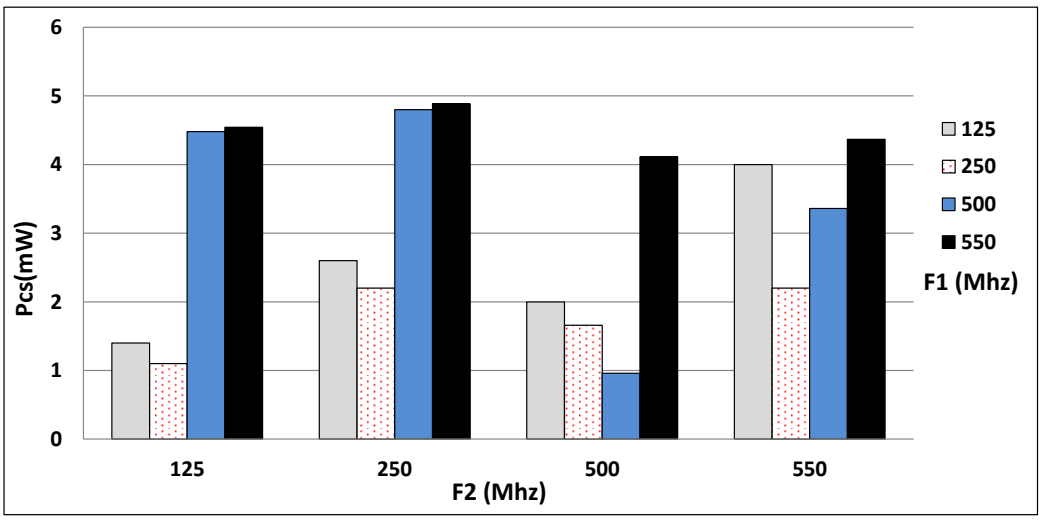


Figure 3.18: Context switch power variation as a function of dynamic CPU frequency scaling

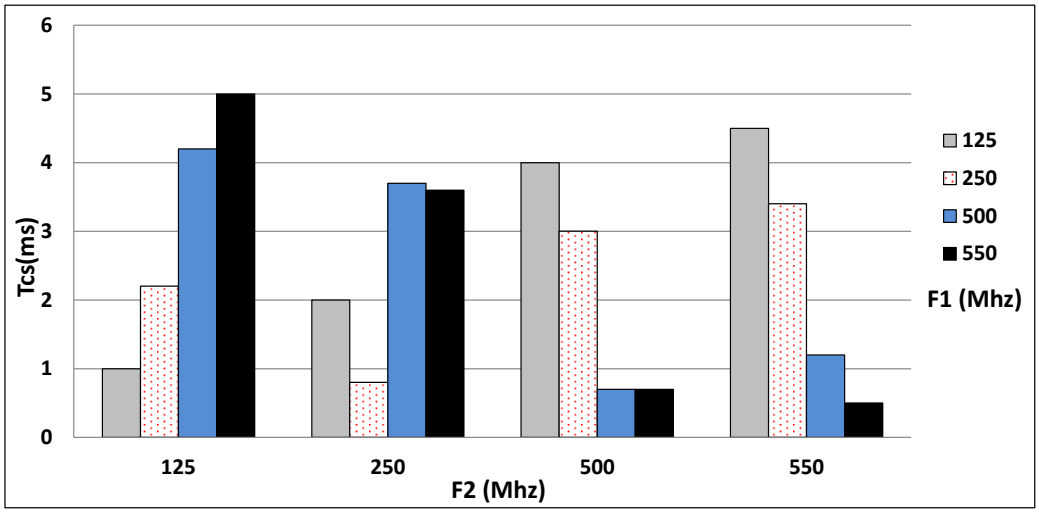


Figure 3.19: Context switch time variation as a function of dynamic CPU frequency scaling

nism, the amount of data shared through the IPC (applicative parameters) and the processor frequency as hardware parameter. Then, we build the power models of the IPC mechanisms. The test programs were developed for three communication mechanisms which are shared memory, named pipes and anonymous pipes. The message length varies from 1B to 8kB, which is the maximum size allowed by the Linux kernel. Communications are performed within the same process to avoid pro-

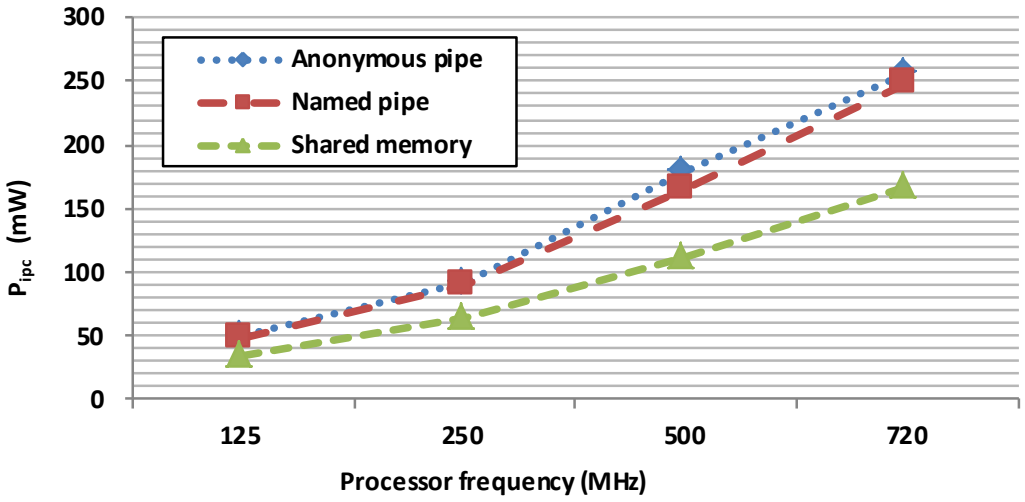


Figure 3.20: IPC power variation as a function of CPU frequency

cess context switching. We have also executed test programs with a high priority to avoid preemptive context switch. Figure 3.20 shows that power consumption P_{ipc} is varying with the processor frequency F. Thus, the power model of IPC mechanisms is:

$$P_{ipc}(F) = (\alpha \times F) + \beta \tag{3.13}$$

Where α and β are coefficients of the model. The unit of P_{ipc} and F is respectively mW and MHz. Power models are presented in table 3.1.
Figure 3.21 depicts the influence of the message size msz on the energy consumption of IPC : E_{ipc}. The energy overhead increases exponentially with the rise of the size of data transmitted.
Equation 3.14 represents the IPC energy model:

$$E_{ipc}(msz) = \lambda \times e^{(\delta \times msz)} \tag{3.14}$$

Where λ and δ are coefficients depending on the message size and the IPC mechanism. The unit of E_{IPC} and msz is respectively nJ and Bytes. Energy models are presented in table 3.2.

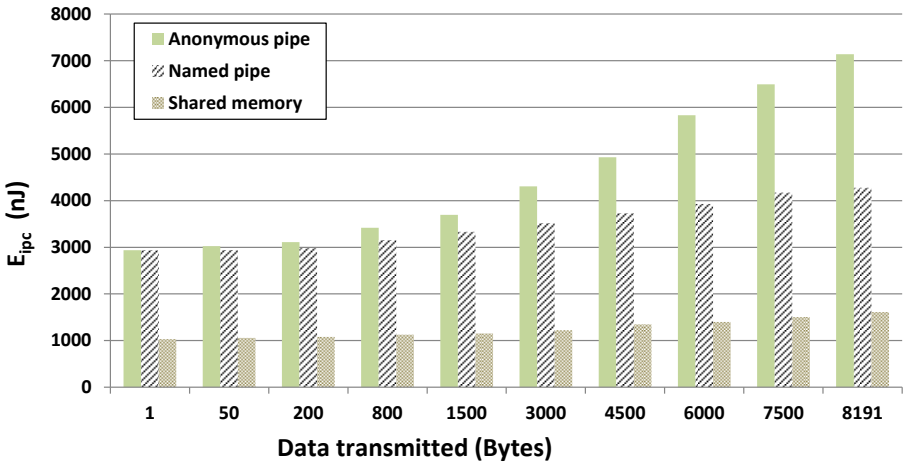


Figure 3.21: IPC energy variation as a function of message size

Table 3.1: Inter-process communication power models according to processor frequency

IPC mechanism	**Power model:** $P_{IPC}(mW)$	**Average error**
Anonymous pipe	$0.347 \times F + 6.474$	0.293%
Named pipe	$0.333 \times F + 4.968$	2.113%
Shared memory	$0.217 \times F + 8.542$	2.27%

Table 3.2: Inter-process communication energy models according to message size

IPC mechanism	**Energy model:** $E_{IPC}(nJ)$	**Average error**
Anonymous pipe	$3068 \times e^{103.9*10^{-6} \times msz}$	2.149%
Named pipe	$3003.8 \times e^{44.96*10^{-6} \times msz}$	1.728%
Shared memory	$1060.9 \times e^{48.9*10^{-6} \times msz}$	1.468%

3.6 Conclusion

In this chapter, the power/energy consumption of embedded OS services were analyzed and and modeled for a specific hardware platform: the OMAP 35X evm board. We proposed a methodology to characterize power/energy overheads of three basic services of the embedded OS: scheduling, context switch and inter-process communication. In addition, the impacts of hardware and software parameters like processor frequency and scheduling policy on energy consumption are studied. Consequently, mathematical models for power and energy consumption are extracted. Next chapter

talks about a high level model of software application, the OS services and hardware platform using an architecture analysis and design language (AADL). Then, AADL and mathematical models of OS services energy consumption will be integrated in a multiprocessor scheduling simulator in order to evaluate the OS energy overhead when using low power techniques.

Chapter 4

High level modeling of embedded system components

Contents

4.1 Exploitation of high level AADL models 60

4.2 Embedded OS functional/non-functional properties and requirements 60

4.3 Architecture modeling languages 61

4.4 Overview of AADL language 63

4.4.1 AADL components 63

4.4.2 Subcomponents 64

4.4.3 Components implementations 65

4.4.4 Components interaction 65

4.4.5 AADL properties, annexes, packages and modes 66

4.4.6 AADL tools 66

4.5 AADL modeling case study 67

4.5.1 H.264 application 67

4.5.2 AADL modeling of system components 68

4.6 Conclusion 77

This chapter introduces a high level modeling of OS services, software and hardware components taking into account the energy consumption aspects. The obtained models will be exploited for calculating OS energy overhead when adapting low power scheduling policies. Also, they will be used for system design exploration and verification of requirements. First, An overview of used modeling language (AADL) is presented. Then, AADL functionalities and tools are exploited to model the OS services and the software application, the H.264 video decoder application. In addition, the communication between OS services and the applicative tasks have been modeled. Furthermore, AADL models of OMAP3 processor and the binding of applicative tasks on the hardware platform components have been proposed.

4.1 Exploitation of high level AADL models

The contribution proposed in this chapter consists in providing a high level AADL models of different hardware and software components, the OS services and the communication between the OS and the applicative tasks. These models take into account the properties of applicative tasks (the deadline, the period etc.), the scheduling policy, the operating points (frequency and voltage) and characteristics of the processor. As showed in figure 4.1, the proposed models will be exploited, in chapter 5, to estimate the OS services energy overhead in order to evaluate the performance of low power scheduling policies. Also, they will be used, in chapter 6, to explore system design and to define and verify various system requirements (OS energy requirements, scheduling requirements etc.).

4.2 Embedded OS functional/non-functional properties and requirements

In embedded systems, non-functional properties or requirements define how a system is supposed to be. They are used to evaluate the system operations. However, functional requirements define a specific behavior of the system. They could be technical details, operations, data manipulation and other various functionalities that specify particular characteristics of a system.
The main non-functional properties of embedded OS are timeliness, dependability and energy consumption [69]. Depending on the kind of deadline, preemption points

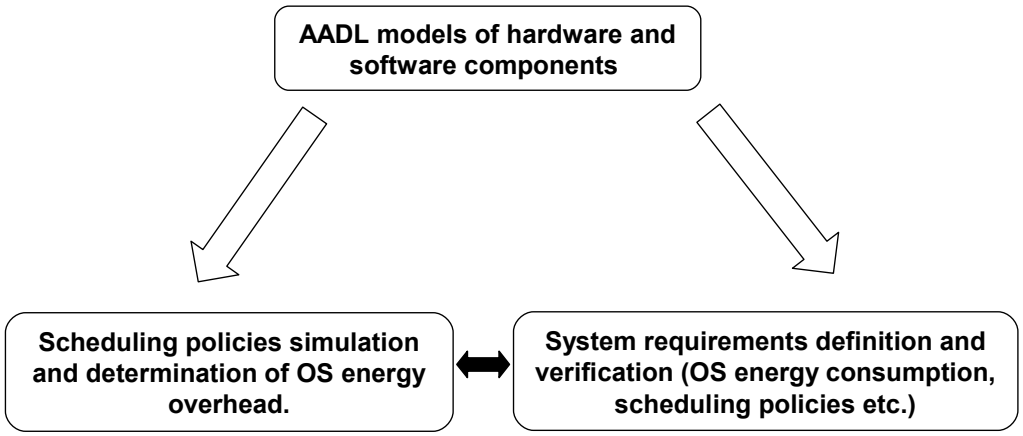


Figure 4.1: From AADL modeling to scheduling policies simulation and system requirements verification

need to be inserted in critical execution paths in order to reduce scheduling latency. The timeliness non-functional issues are the existence, locality, and frequency of a preemption point.
The dependability non-functional property refers to the system trustworthiness providing a service that can be justifiably trusted. The dependability is *the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s)* [26]. It encompasses aspects of reliability, availability, safety, security, survivability and maintainability. In fact, these aspects rely on hardware-supported protection and isolation in order to limit fault propagation.
The last non-functional requirement is the energy which a scarce resource in embedded systems, especially the battery operated embedded systems such as mobiles. As demonstrated in chapter 3, the embedded OS energy overhead is important and varies with different hardware and software parameters. Consequently, we are interested in modeling this non functional property for embedded OS services.
In this work, we aim at building a model of the functional/non-functional properties and requirements. So, choosing the adequate modeling language, to analyze and verify these embedded systems properties, is necessary. This choice is justified in next section.

4.3 Architecture modeling languages

In order to describe and analyze functional/non-functional properties and requirements of a system, various modeling languages are used. The Object Management Group (OMG) [17] standardizes the Unified Modeling Language (UML) [18], a widespread modeling language including a set of graphic notation techniques to provide concepts and model the architecture behavior and the deployment of software systems in object-oriented or component-based paradigms.
The UML language can be extended through profiles, accommodating domain specific modeling concepts. For example, SysML [4] is a profile to describe system engineering applications. Non-functional properties and requirements are harder to describe using UML. An extension of UML called MARTE [108] improves UML functionalities to enhance the modeling and analysis of Real-Time and Embedded systems. The MARTE language takes into account different aspects such as schedulability, performance and time [78]. This language addresses new requirements: specification of both software and hardware model aspects, separated abstract models of software applications and hardware platforms and modeling various domains of time and non functional properties [30].
The Architecture Description Languages (ADLs) target to model both functional and nonfunctional properties of system architectures.

The (ADL) language generates different executable models with simulator, compiler and hardware configuration. The generated models enable various design automation tasks including exploration, simulation, compilation, synthesis, test generation and validation. Furthermore, (ADL) language is exploited to design both software and hardware architectures. It analyzes and models the software applications architectures [41] by capturing behavioral specifications of applicative tasks and their interactions. Also, (ADL) language describes the hardware platform. It models the different modules of the platform and their connectivity.
Recently, various (ADL)s have been proposed and exploited for modeling the system functional/non functional properties [70].
From these proposals, the Architecture Analysis and Design Language (AADL) [77], developed by the Society of Automotive Engineers (SAE), has received increasing interest from mission-critical applications development industries. The AADL standard models the applications and hardware platforms and describes the deployment of applicative tasks on hardware components. The modeling is performed using textual and graphic notations with precise semantics. This language analyzes and models the functional and non functional requirements and properties of embedded systems. The (SAE) generates, from the AADL model, textual files with interchange text format (XML) that supports the exchange of AADL models between different subcontractors, integrators and agencies. Additionally, the AADL standard is extensible with analysis approaches to evaluate properties such as schedulability, performance and power/energy consumption. The standard of this language was proposed in 2004 and functionalities were published in 2006 for graphical notation, error modeling, standard meta-model and programming language guidelines. Moreover, this language is supported by commercial and open source tool solutions: the Open Source AADL Tool Environment (OSATE) [106].
For these reasons, under the OPEN-PEOPLE project, different academic and industrial partners choose AADL as modeling language.
Various works use the AADL to model system architecture and verify its constraints. In [88], the authors use the AADL language to define memory architectures, and then verify rules in order to assess that the memory is correctly dimensioned. They model memory requirements (such as layout or size) and then validate them on a case-study using the VxWorks real-time kernel.
Also, an AADL simulation tool has been proposed in [112] to design and analyze software and hardware architectures for real-time embedded systems. This tool supports the performance analysis of the AADL specification throughout the refinement process from the initial system architecture until the complete, detailed application and execution platform are developed. AADL language is used to verify the initial timing constraints during the complete design process.

4.4 Overview of AADL language

To describe AADL models of different features of an embedded system, various representations are available for the AADL users:

- **Graphical representation:** This kind of representation is used to show an overview of the system and the interaction between the application and different hardware devices.
- **Textual representation:** The text format is used to refine and detail the model entities.
- **XML format representation (AAXL file):** The XML file is also used for AADL modeling to facilitate the interoperability between different tools.

In this section, we present an overview of AADL specification of embedded systems by showing the different available software, hardware and hybrid components. Also, implementations and properties of these components will be detailed.

4.4.1 AADL components

To model complex embedded systems, AADL provides three distinct sets of component categories:

4.4.1.1 Software components

In order to describe the applicative tasks, AADL model includes various software components:

- **Thread:** it is the smallest sequence of routines that can be scheduled by an OS. A thread represents a unit of concurrent execution.
- **Process:** represents a protected address space. A process should include one or more threads.
- **Thread group:** the hierarchy of thread group is used to organize threads within a process in the same block.
- **Data:** this software component includes application data types and different data component implementations.
- **Subprogram:** this component is a sequential code that could be called for execution. A subprogram could call other subprograms and communicate with them through specific parameters and data access features.

The AADL language assigns to each software component a graphical symbol. Figure 4.2 depicts different graphical representations of software components.

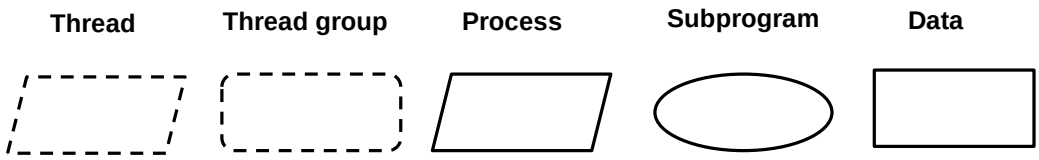


Figure 4.2: Graphical representations of software AADL components

4.4.1.2 Hardware components

Various hardware components are available for AADL users in order to model different units of hardware platforms and to represent system's computational and interfacing resources. These hardware platform components are:

- **Processor:** represents the main hardware unit of the platform that runs and schedules the threads.
- **Memory:** represents the hardware devices allowing the storage of data and routines.
- **Bus:** this component ensures the interconnection between different hardware parts of the system.
- **Device:** represents different entities of the external environment such as peripheral devices.

4.4.1.3 Generic components

The generic or composite component, called *system* in AADL modeling, is used to model entities consisting of both hardware and software components. The system component encapsulates hardware devices, such as peripheral device or processor, the software application tasks and the mapping of software code on hardware components. Generic components add hierarchy in the modeling; they are at the higher level in this hierarchy. Figure 4.3 represents the different AADL graphical representations of execution platform and generic components.

4.4.2 Subcomponents

A subcomponent represents a component instance that defines the category, type and specifications of an AADL component. Different subcomponents of each AADL component are presented in table 4.1.

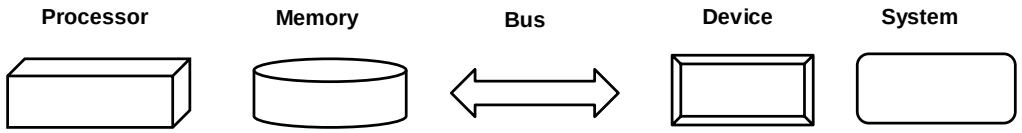


Figure 4.3: Graphical representations of hardware and generic AADL components

Table 4.1: AADL subcomponents

Components	Subcomponents
System	Data, process, subprogram, processor, memory, bus, device, system
Thread	Data
Thread group	Thread, thread group, data
Process	Thread, thread group, data
Processor	Memory, bus
Memory	Memory, bus
Bus	None
Data	Data
Device	None
Subprogram	None

4.4.3 Components implementations

The component implementation describes the internal structure of different AADL components. It specifies the set of subcomponents and details interactions between their features. These interactions are established through connections, calls, bindings and mapping of software components on the hardware platform. The component implementation defines different modes representing operational states and component properties [52].

4.4.4 Components interaction

To ensure the communication between AADL components, the AADL developers provide various interfaces or features. A component interface consists of directional flow through:
- Data ports for unqueued state data.
- Event data ports for queued message data.
- Event ports for asynchronous events.
- Subprogram calls.
- Explicit access to data components.

4.4.5 AADL properties, annexes, packages and modes

AADL language not only describe the architecture and interconnections between components, but also the behavior of different components. It specifies the components characteristics using the properties, annexes, packages and modes.

4.4.5.1 AADL properties

An AADL property provides information about an AADL specification element. The timing characteristics of different applications tasks, such as the deadline, the worst-case execution time and the period, are defined through AADL properties. Furthermore, AADL properties include the source code and routines of AADL modeled applicative software components and they specify the constraints for binding threads to processors, processes to memories and connections to busses. Properties are declared in named property sets. Property set declarations allow the addition of properties to the core of AADL property set.

4.4.5.2 AADL annexes

The AADL annexes enrich the architecture description using specific languages such as Object Constraint Language (OCL). Many annexes have been defined by AADL developers, for example, the error-model annex that specifies fault and propagation concerns and the data-model annex that describes the modeling of specific data constraint with AADL.

4.4.5.3 AADL packages

The libraries of AADL components are defined in AADL packages. These packages organize the import of component declarations.

4.4.5.4 AADL modes

AADL modes are the operational states of software, hardware and compositional components in the modeled system.

4.4.6 AADL tools

Various tools are available for system modeling and analysis with the AADL language. For instance, Ocarina tool is used for optimization and analysis of AADL models [58]. This tool achieve semantic analysis, schedulability analysis and checks the behavior of the model by transforming the AADL model to a Petri network and

performing formal verification. Ocarina tool is able to generate automatically a code from AADL models to C code, ADA code and ARINC653 compliant systems. Also, Cheddar [103] is a free real time scheduling tool. It is designed for checking task temporal constraints of a real time application/system written in the AADL or with a Cheddar proprietary language. Cheddar is not used in this work because it supports only monoprocessor platforms.
In the context of the OPEN-PEOPLE project, we use a set of ECLIPSE [46] based tools. Also, the OSATE is used as textual and graphical modeling tool. It is defined as a set of plug-ins on top of the open source Eclipse platform. The set of plug-ins provides a tool set for front-end processing of AADL models. These models can be maintained as textual AADL files or as XML based AADL model files. Also, the ADELE tool is used in this work [107]. It addresses shortcomings in the OSATE graphical editor by providing a new AADL editor, with a new graphical layer.

4.5 AADL modeling case study

In this section, we present AADL models of different embedded system components. The use case software application, the H.264 video decoder, is detailed below.

4.5.1 H.264 application

The H.264 video decoder application is taken as main use case application. It is a high quality video compression algorithm relying on several efficient strategies extracting spatial (within a frame) and temporal dependencies (between frames). This application is characterized by a flexible coding, high compression and high quality resolution. Moreover, it is a promising standard for embedded devices.
The main steps of the H.264 decoding process consist in the following: first, a compressed bit stream coming from the Network application layer (NAL), which formats the representation of the video and provides header information in a manner appropriate for conveyance by particular transport layers, is received at the input of the decoder. Then, the entropy decoded bloc begins with decoding the slice header and then it decodes the other parameters. The decoded data are entropy decoded and sorted to produce a set of quantized coefficients. These coefficients are then inversely quantized and transformed. Thereafter, the data obtained are added to the predicted data from the previous frames depending upon the header information. Finally, the original block is obtained after the de-blocking filter to compensate the block artifacts effect.
The H.264 video decoder application can be broken down into various tasks sets corresponding to different types of parallelization. In our experiments, we use the slices

version proposed by Thales Group (France) [10] in the context of French national project PHERMA (Parallel Heterogeneous Energy efficient real-time Multiprocessor Architecture) [5].
The main characteristic of this version is that the algorithm is parallelized on the slices of the frame as illustrated in figure 4.4 from this diagram; For this version, it is considered that frames are made up of 4 slices. Since slices inside a frame can be computed independently, therefore, one task is assigned for each slice to be computed. Thus, four tasks, named *slice_processing*, can run simultaneously in this version. There are some synchronizations required between tasks that must be handled to ensure a proper processing without data corruption. These synchronizations are handled through the task named $SYNC$. At the beginning of each new frame, tasks can access only sequentially to the input data buffer. Therefore, there is a slight overhead in the real beginning of each start up of the task named *slice*. This behavior is due to the access of shared resource which is protected by a semaphore. Due to temporal dependencies between frames, it is not possible to compute the next frame if the previous one has not been completely decoded. Thus, at the end of each slice computation, tasks need to be resynchronized using task named $SYNC$. As a result, input data must be present and the previous frame must be decoded at the start of decoding a new frame. Hence, we have four types of tasks. First, we start with the NEW_FRAME task (T_1) that can access only sequentially to the input data buffer. Therefore, the $NAL_DISPATCH$ task (T_2), which provides access to a shared resource and is protected by a semaphore, starts execution. Then, $SLICE_PROCESSING$ tasks (T_3, T_4, T_5 and T_6) are launched simultaneously. Due to temporal dependencies between frames, it is not possible to compute the next frame if the previous one has not been completely decoded. Thus, at the end of each slice computation, tasks need to be resynchronized using the $SYNC$ task before running the $REBUILD_FRAME$ (T_7) task.
Hence, H.264 slices version, comprising seven periodic tasks as shown in table 4.2, is used as use case application. All values are given at maximum frequency of OMAP 3 processor (i.e., 720-MHz).

4.5.2 AADL modeling of system components

Using AADL, the properties of the system architecture, including the application's tasks and the hardware platform, are modeled.

4.5.2.1 H.264 application software tasks AADL modeling

The software tasks are modeled using the "Thread" component. The model is divided into two parts: the features, that define the component interface and its communication ports, and the properties, that define the task's period and its acti-

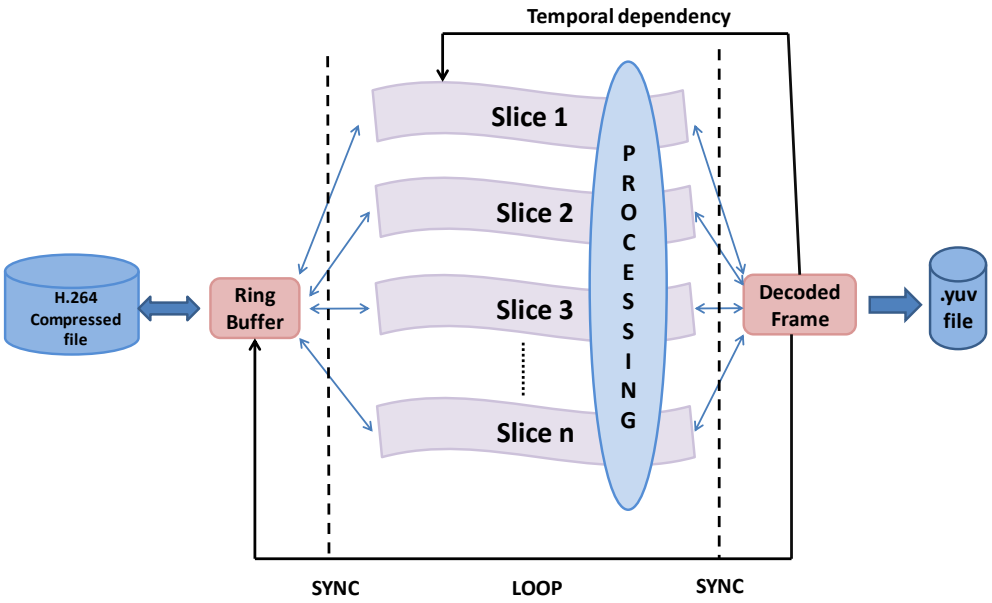


Figure 4.4: Block diagram of H.264 decoding scheme slices version

Table 4.2: H.264 video decoder application tasks features

Task name	WCET (ms)	BCET (ms)	Period (ms)	Deadline (ms)	Activation date (ms)
$New_frame(T1)$	1	1	19	19	0
$Nal_dispatch(T2)$	2	1	5	5	0
$Slice1_processing(T3)$	42	21	66	66	0
$Slice2_processing(T4)$	42	21	66	66	1
$Slice3_processing$ (T5)	42	21	66	66	2
$Slice4_processing$ (T6)	42	21	66	66	3
$Rebuild_frame(T7)$	2	1	66	66	66

vation protocol. The execution times of different tasks will be modeled when binding the application tasks on the hardware platform. For instance, figure 4.5 depicts the AADL model of the second task of the application $NAL_DISPATCH$: T_2.

```
thread Nal_dispatch
features
IN0: in event data port donnee.impl;
OUT1: out event data port donnee.impl;
OUT2: out event data port donnee.impl;
OUT3: out event data port donnee.impl;
OUT4: out event data port donnee.impl;
properties
Dispatch_Protocol => Periodic;
Period => 5 Ms;
end Nal_dispatch;
```

Figure 4.5: Thread $NAL_DISPATCH$ AADL model

4.5.2.2 AADL modeling of OS services

The AADL implementation of each OS service specifies its properties, such as the periodicity, and its features, mainly, its communication ports. The OS services AADL model is presented in figure 4.6. To model the different OS services studied and to facilitate the communication with applicative tasks, we need to gather these services in the same unit. For this reason, we use the "Thread group" component that includes three "Thread"s representing the context switch, the inter-process communication and the scheduling services. Figure 4.7 shows the structure of OS services AADL unit.

4.5.2.3 AADL modeling of communication between OS services and applicative tasks

When switching from one task to another, the OS routines are called. To ensure the communication and message passing between the OS services and the applicative tasks, the event/data ports are used. As showed in figure 4.8, the software components are linked by event and data connections. The AADL graphical representation of different software tasks and their interactions with the OS services are showed in figure 4.9.

```
--------- OS Services threads-------

  thread Scheduling
    features
      IN0: in event data port donnee.impl;
      OUT0: out event data port donnee.impl;
  end Scheduling;

  thread implementation Scheduling.impl
    properties
      Dispatch_Protocol => Periodic;
      Period => 1 Ms;
      POSIX_Properties::POSIX_scheduling_policy => SCHED_OTHER;
  end Scheduling.impl;

  thread IPC
    features
      IN0: in event data port donnee.impl;
      OUT0: out event data port donnee.impl;
  end IPC;

  thread implementation IPC.impl
    properties
      POSIX_Properties::POSIX_Memory_policy => SHARED_MEMORY;
      Dispatch_Protocol => Periodic;
      Period => 1 Ms;
  end IPC.impl;

  thread Context_Switch
    features
      IN0: in event data port donnee.impl;
      OUT0: out event data port donnee.impl;
  end Context_Switch;

  thread implementation Context_Switch.impl
    properties
      Dispatch_Protocol => Periodic;
      Period => 1 Ms;
  end Context_Switch.impl;
```

Figure 4.6: OS services AADL model

```
--------- OS Services Thread Group -------

thread group OS_services
 features
   IN0: in event data port donnee.impl;
   OUT0: out event data port donnee.impl;
end OS_services;

thread group implementation OS_Services.impl
   subcomponents
         SCHED: thread Scheduling.impl;
         IPC: thread IPC.impl;
         CS: thread Context_Switch.impl;
   connections
         event data port IN0 -> SCHED.IN0;
         event data port SCHED.OUT0 -> IPC.IN0;
         event data port IPC.OUT0 -> CS.IN0;
         event data port CS.OUT0 -> OUT0;
end OS_Services.impl;
```

Figure 4.7: Thread group of OS services

4.5.2.4 AADL modeling of OMAP 3 processor

To model the OMAP 3 processor, we use various AADL hardware components such as "processor", "bus" and "memory" components. The bus is used for the communication between the processor core and its memory. When implementing the processor model, as depicted by figure 4.10, the AADL memory component is used to define the internal cache memory of the processor. Also, many characteristics of the processor are specified such as its operating point (Running frequency and voltage), its idle and running power consumption and the scheduling policy used.

4.5.2.5 AADL modeling of the tasks binding on hardware components

To provide a complete system specification, applicative tasks should be bound to appropriate execution platform components. This software/hardware binding is modeled using property associations called binding properties. The AADL modes are exploited to represent different system states. In the proposed model, each system mode is mainly characterized by the processor running frequency, voltage and power consumption in idle and running states. Many properties are linked to each system mode such as the power consumption in idle and running states and the CPU core voltage.
When switching from one mode to another, the system active components and con-

```
process basic_process
        properties
        Threads_Number => 7 threads;
end basic_process;

process implementation basic_process.impl
  subcomponents
        NEW_FRAME: thread New_frame.impl;
        NAL_DISPATCH: thread Nal_dispatch.impl;
        SLICE1_PROCESSING: thread Slice1_processing.impl;
        SLICE2_PROCESSING: thread Slice2_processing.impl;
        SLICE3_PROCESSING: thread Slice3_processing.impl;
        SLICE4_PROCESSING: thread Slice4_processing.impl;
        REBUILD_FRAME: thread Rebuild_frame.impl;
        CALL1_OS, CALL2_OS, CALL3_OS : thread group OS_Services.impl;
  connections
        event data port NEW_FRAME.OUT0 -> CALL1_OS.IN0;
        event data port CALL1_OS.OUT0 -> NAL_DISPATCH.IN0;
        event data port NAL_DISPATCH.OUT0 -> CALL2_OS.IN0;
        event data port CALL2_OS.OUT1 -> SLICE1_PROCESSING.IN0;
        event data port CALL2_OS.OUT2 -> SLICE2_PROCESSING.IN0;
        event data port CALL2_OS.OUT3 -> SLICE3_PROCESSING.IN0;
        event data port CALL2_OS.OUT4 -> SLICE4_PROCESSING.IN0;
        event data port SLICE1_PROCESSING.OUT0 -> CALL3_OS.IN1;
        event data port SLICE2_PROCESSING.OUT0 -> CALL3_OS.IN2;
        event data port SLICE3_PROCESSING.OUT0 -> CALL3_OS.IN3;
        event data port SLICE4_PROCESSING.OUT0 -> CALL3_OS.IN4;
        event data port CALL3_OS.OUT0 -> REBUILD_FRAME.IN0;
end basic_process.impl;
```

Figure 4.8: AADL modeling of the communication between application and OS services

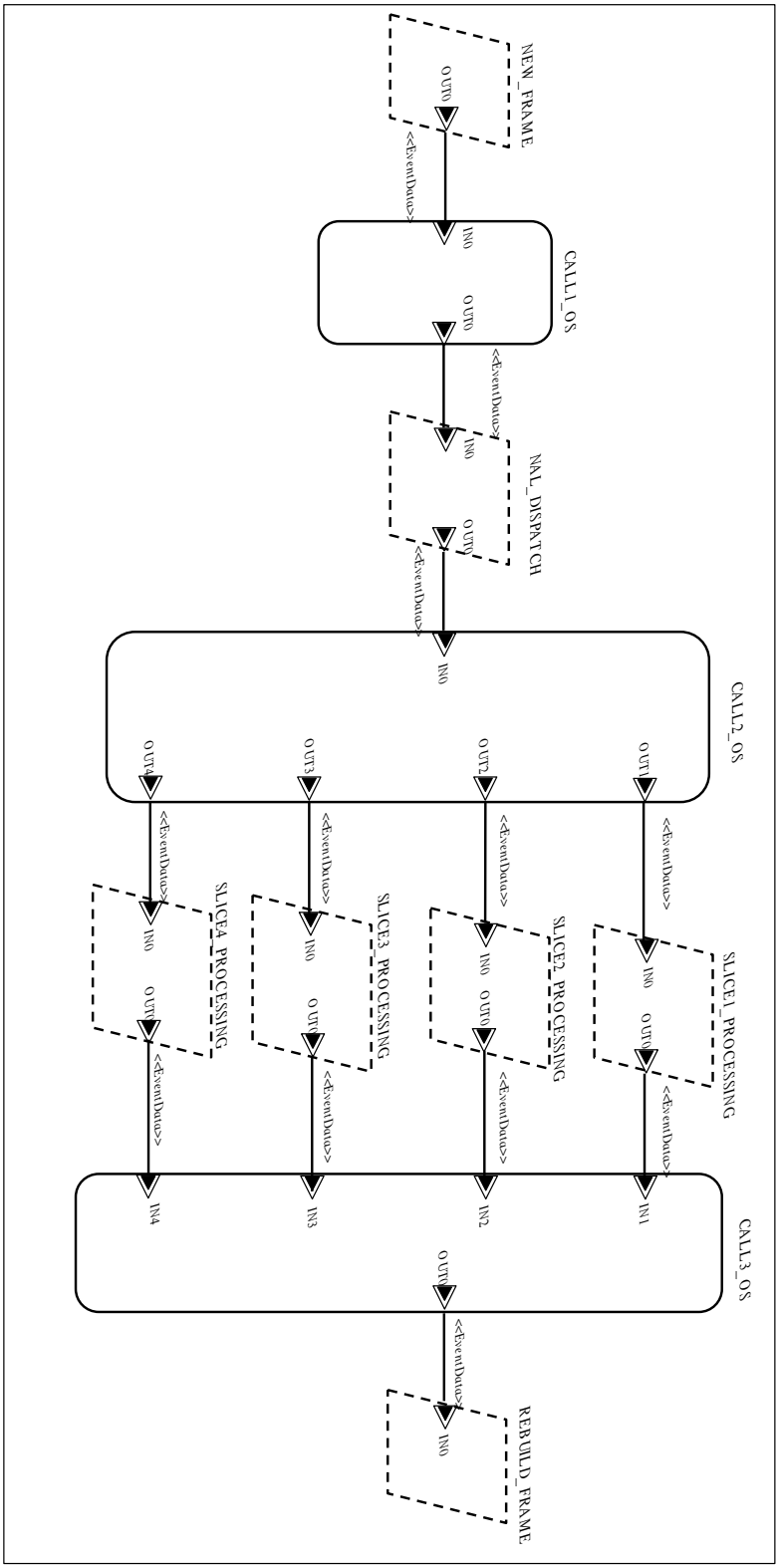


Figure 4.9: AADL Graphical representation of the communication between tasks and OS services using the ADELE tool

```
processor OMAP_3530
 features
   bus_access: requires bus access Bus2.impl;
end OMAP_3530;

processor implementation OMAP_3530.impl
 subcomponents
   CACHE: memory Memory1;
 properties
   POSIX_Properties::POSIX_scheduling_policy => SCHED_RR;
   Power_Properties::Power_Consumption => 57.0 mW;
   Power_Properties::Power_Idle => 4.0 mW;
   Basic_OP_Properties::VDD => 1.0 V;
   Basic_OP_Properties::Frequency => 125.0 MHz;
end OMAP_3530.impl;
```

Figure 4.10: AADL modeling of the OMAP3530 processor

nections between them change due to the processor frequency variation. The AADL event ports are used to ensure the system mode change, the activation and deactivation of system parts, when switching from one frequency to another. Figure 4.11 shows the different system modes, used when binding the applicative tasks on the processor, for different operating points of the processor: 125 MHz, 250 MHz, 500 MHz and 720 MHz. It also depicts the event ports allowing the switching from one mode to another.

Figure 4.12 details the binding properties. The property type "reference" allows a property value to refer to a model element according to the containment hierarchy. The *Allowed_ Processor_ Binding* declaration references modeled processor in the system hierarchy. This property association restricts the binding to processors of type OMAP 3 and is applied to the software part of the system, the H.264 Application. It also specifies the system execution mode. Besides, the *Actual_ Memory_ Binding* property association defines the memory component to which code and routines are bound.

The reference properties should be declared high enough in the system hierarchy in order to point to the desired component in the system hierarchy.

```
system Global_Binding
 features
   Frequency_at_125_MHZ: in event port;
   Frequency_at_250_MHZ: in event port;
   Frequency_at_500_MHZ: in event port;
   Frequency_at_720_MHZ: in event port;
end Global_Binding;

system implementation Global_Binding.impl
 subcomponents
   H264_application: system ApplicationH264::H264.impl;
   Execution_platform: system Multiprocessor_plat::main.impl1;
   SRAM: memory Multiprocessor_plat::Memory1;
 modes
   conf0: initial mode ;
   conf1: mode {
         Basic_OP_Properties::Frequency => 125.0 MHz;
         Power_Properties::Power_Consumption => 57.0 mW;
         Power_Properties::Power_Idle => 4.0 mW;
         Basic_OP_Properties::VDD => 1.0 V;
         };
   conf2: mode {
         Basic_OP_Properties::Frequency => 250.0 MHz;
         Power_Properties::Power_Consumption => 130.0 mW;
         Power_Properties::Power_Idle => 7.0 mW;
         Basic_OP_Properties::VDD => 1.1 V;
         };
   conf3: mode {
         Basic_OP_Properties::Frequency => 500.0 MHz;
         Power_Properties::Power_Consumption => 303.0 mW;
         Power_Properties::Power_Idle => 16.0 mW;
         Basic_OP_Properties::VDD => 1.3 V;
         };
   conf4: mode {
         Basic_OP_Properties::Frequency => 720.0 MHz;
         Power_Properties::Power_Consumption => 550.0 mW;
         Power_Properties::Power_Idle => 28.0 mW;
         Basic_OP_Properties::VDD => 1.35 V;
         };
   conf0 -[ Frequency_at_125_MHZ ]-> conf1;
   conf0 -[ Frequency_at_250_MHZ ]-> conf2;
   conf0 -[ Frequency_at_500_MHZ ]-> conf3;
   conf0 -[ Frequency_at_720_MHZ ]-> conf4;
```

Figure 4.11: AADL implementation of system modes and events

```
Properties
Allowed_Processor_Binding => ( reference
Execution_platform.cpu1, reference
Execution_platform.cpu2, reference
Execution_platform.cpu3, reference
Execution_platform.cpu4, reference
Execution_platform.cpu5) applies to
H264_application.pr;
Actual_Memory_Binding => reference sram applies to
H264_application.pr;
```

Figure 4.12: AADL model of software application on hardware platform

4.6 Conclusion

In this chapter, we have presented an overview of AADL language. The AADL functionalities and tools have been exploited to model the OS services and the software application, the H.264 video decoder application. In addition, the communication between OS services and the applicative tasks have been modeled. Furthermore, AADL models of OMAP3 processor and the binding of applicative tasks on the hardware platform components have been proposed. The implemented AADL models will be used in next chapter for the determination of OS services energy overhead when adapting low power scheduling policies.

Chapter 5

Embedded OS service's models integration in the system level design flow

Contents

5.1 Models integration in multiprocessor scheduling simulation tool . . . 80
5.1.1 STORM tool . . . 80
5.1.2 The proposed approach . . . 81
5.2 Low power scheduling policies . . . 83
5.2.1 The AsDPM scheduling policy: . . . 84
5.2.2 The DSF scheduling policy: . . . 87
5.3 Embedded OS services energy overhead: . . . 88
5.3.1 Fixed frequency case: . . . 89
5.3.2 Dynamic frequency case: . . . 89
5.4 Experimental results: . . . 91
5.5 Conclusion . . . 94

Thanks to the significant evolution in processor technology over the last few years, processors with variable voltages and frequencies are now available, they adapt low power and energy scheduling policies to minimize the energy consumption. Reduction in supply voltage requires reduction in operating frequency. To ensure a high level of energy and power optimization, several studies and techniques have been proposed for the exploration of scheduling policy and dynamic Voltage/Frequency management. For instance, the Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS) techniques are used to reduce the power and energy consumption. In this chapter, the energy overhead of studied OS services is evaluated when using an instance of DPM and DVFS low power techniques: the AsDPM and DSF scheduling policies. In fact, the operating system services's models are integrated at system level using multiprocessor scheduling simulator (STORM). Also, a general flow, consisting mainly in generating from the AADL model a file

used as input to the STORM simulator, and calculating the OS energy overhead, is proposed in this chapter.

5.1 Models integration in multiprocessor scheduling simulation tool

The integration of OS services energy models in power/energy estimation tools is necessary to achieve estimations at system level and to quantify the power/energy overhead of embedded OS services. In this work, energy estimation is targeting the system design including software and hardware components. Hence, the OS power/energy mathematical and AADL models, developed respectively in chapter 3 and 4, will be integrated in a simulation tool.
We present in the remaining of this section the simulation tool used to integrate OS services energy models at system level. Then, we introduce the proposed methodology of energy models integration at system level.

5.1.1 STORM tool

To simulate the execution of application and extract the OS energy overhead when using low power scheduling policies, we use STORM (Simulation TOol for Real-time Multiprocessor Scheduling) simulator [9]. This tool is a java-based simulator for multiprocessor scheduling algorithms developed by IRCCyN [83] under the French national project PHERMA [5].
The main functionality of STORM tool is evaluating of performance and energy consumption efficiency of software applications and hardware platforms. This simulation tool allows the implementation of different scheduling policies on multiprocessor architectures with homogeneous or heterogeneous processors. This simulator takes into account the architecture and different components of hardware platforms, such as the multi-core design and memory architecture (L1 and L2 caches), and low power consumption policies, particularly DPM and DVFS techniques. STORM is characterized by high flexibility, so that the user could add many simulation entities, and by portability: the possibility of running on various operating systems.
As shown in figure 5.1, the inputs of this tool is the specifications of the hardware and software architectures together with the scheduling policy; it simulates the system behavior using all the characteristics (task execution time, processor functioning conditions, etc.) in order to obtain the chronological track of different scheduling events that occurred at run time, compute various real-time metrics and analyze the system behavior and performances from various point of views.
Different tasks, data links and processor entities are specified in XML input file, and they are automatically instantiated from the library components. These libraries

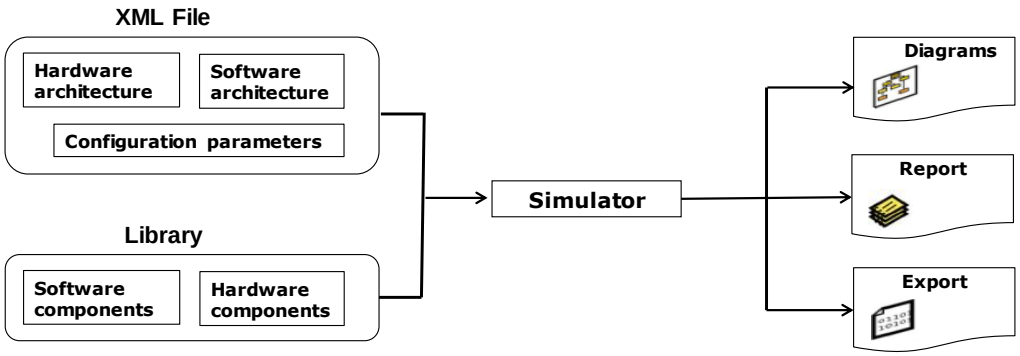


Figure 5.1: STORM simulator input and output file system [9]

define the task characteristics: recurrence, periodicity, aperiodicity, etc. An example of XML file is showed in figure 5.2. It represents the set of used processors and software application tasks with various characteristics such as the best case execution time (BCET), the period and deadline.

5.1.2 The proposed approach

As demonstrated in figure 5.3, the proposed approach of OS services integration at system level revolves around three focal concepts: AADL Modeling, code transformation and energy/power estimation.

The AADL modeling step is mainly performed in last chapter. Different AADL models are integrated in this flow. In fact, we rely on the platform model that contains all the components and connections instances of the application. Also, we perform the implementation of various components instances, found in the AADL models library. Furthermore, AADL model is exploited to describe the hardware of the physical target platform: the processor, the memory, and the bus entity which are necessary to processes and threads execution. In the proposed approach, we take into account the intra-task properties, such as the deadline and worst case execution time, and the inter-task aspects, such as the events and inter-process communication, in order to define the binding properties that are necessary to the deployment of the application's tasks and embedded OS services on the target platform. Using the textual and graphical modeling tool OSATE, we automatically generate the corresponding textual deployment file: the AADL model is mapped to an XML file. As a result, simulated outputs can be computed as: either user readable in the form of diagrams or reports, or machine readable intended for a subsequent analysis tool. The user interacts with STORM through a user-friendly graphical interface which is composed of command and display windows. The XML file generated from the AADL model having the extension ".aaxl" is not recognized by the STORM simu-

```
<SIMULATION duration="10000" precision="1">


<SCHED className="EDF_P_Scheduler" quantum="1"> </SCHED>

<CPUS>

 <CPU className="storm.Processors.OmapProcessor" name="CPU A" id="1"></CPU>
 <CPU className="storm.Processors.OmapProcessor" name="CPU B" id="2"></CPU>

</CPUS>

<TASKS>

  <TASK className="storm.Tasks.PTask_NAM_A" name="NEW_FRAME" id="1"
activationDate="0" WCET="1" BCET="1" period="19" deadline="19"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="NAL_DISPATCH" id="2"
activationDate="0" WCET="2" BCET="1" period="5" deadline="5"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="SLICE1_PROCESSING" id="3"
activationDate="0" WCET="42" BCET="21" period="66" deadline="66"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="SLICE2_PROCESSING" id="4"
activationDate="1" WCET="42" BCET="21" period="66" deadline="66"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="SLICE3_PROCESSING" id="5"
activationDate="2" WCET="42" BCET="21" period="66" deadline="66"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="SLICE4_PROCESSING" id="6"
activationDate="3" WCET="42" BCET="21" period="66" deadline="66"> </TASK>


  <TASK className="storm.Tasks.PTask_NAM_A" name="REBUILD_FRAME" id="7"
activationDate="66" WCET="2" BCET="1" period="66" deadline="66"> </TASK>

</TASKS>
</SIMULATION>
```

Figure 5.2: Example of STORM input XML file

lator. For this reason, in the code transformation step, we adapt the file generated to the simulator structure by parsing existing file ".aaxl " and extracting the data needed to generate the input file of the simulator. To extract the required data from the "aaxl" file, we use the java API JDOM [16] which allows us to manipulate and output XML data from Java code. Consequently, we can read and write XML data without the complex and memory-consumptive options that current API offerings provide. Because JDOM uses the Java Collections API to manage a tree data structure, we transform the "aaxl" file to a JDOM tree. Then, we extract each data by walking the tree and iterating the document as showed in algorithm 2.

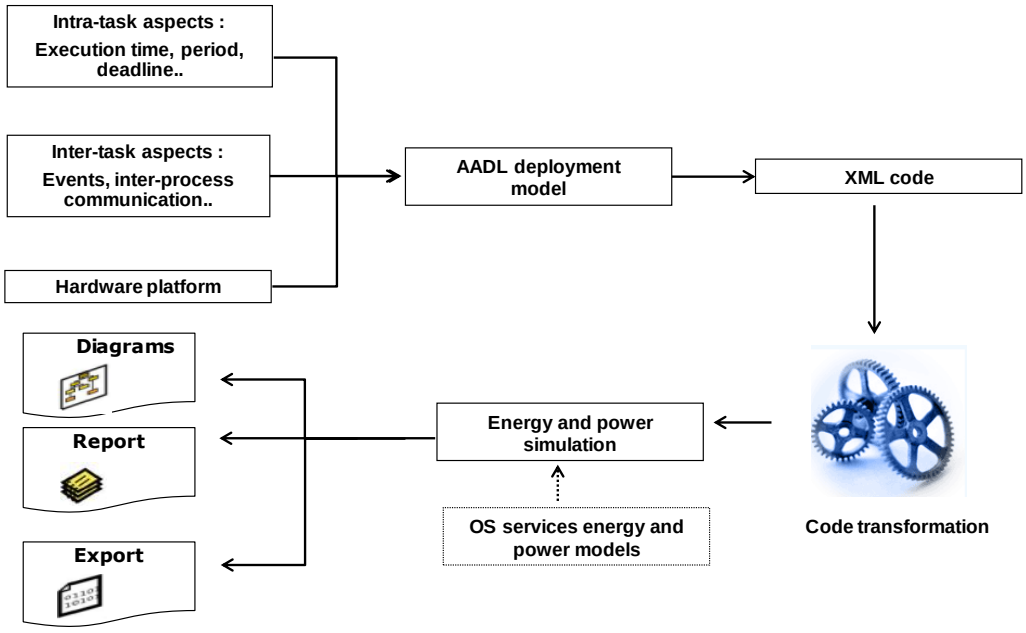


Figure 5.3: Os power and energy models integration in the system level design flow

For instance, the code of task period extraction is depicted by figure 5.4. In the re-

Algorithm 2 Data extraction from the "aaxl" file

```
1: Create a list of the tree's nodes
2: Create an iterator "i" for the list
3: while "i" has a next element in the tree do
4:     assign  j = next element of "i"
5:     Extract the required property;
```

maining of this chapter, taking as use case the H.264 video decoder application, the energy consumption of the OS services will be determined when adapting specific low power techniques, presented in next section.

5.2 Low power scheduling policies

In this section, we present the low power techniques used to evaluate the performance of embedded OS services: the DSF and AsDPM techniques. These techniques work in conjunction with global Earliest Deadline First (EDF) scheduling algorithm. On single-processor, under the EDF scheduling policy, at every time instant, the task that has the smallest deadline is selected for execution on the sole processor. (EDF)

```
static void period (String Attribute2)
 {
   List thread = racine.getChildren("threadimpl") ;
   Iterator i = thread.iterator();

    While (i.hasNext())

    {
       Element threadCourant=(Element) i.next();
       List PropAssociation=threadCourant.getChild("properties").getChildren("propertyAssociation");

       Iterator j=PropAssociation.iterator();

      While (j.hasNext())

      {
        Element PropertyAssociation=(Element) j.next();

        String test= PropertyAssociation.getAttributeValue("propertyDefinition").toString();

            If (test.equals(Attribute2))
            {

ThreadCourant.setAttribute("period",PropertyAssociation.getChild("PropertyValue").getAttributeValue("value"));

            }
       }
      }
}
```

Figure 5.4: Task period extraction

is optimal scheduling algorithm for single-processor systems [36, 67]. Nevertheless, when more processors are added to the system, (EDF) suffers from sub-optimality. The used low power scheduling policies aim to verify (EDF) scheduling constraints and reduce the energy consumption of multiprocessor hardware platform when running the application tasks.

5.2.1 The AsDPM scheduling policy:

The dynamic power management (DPM) is an efficient technique for embedded systems energy reduction [75, 96]. When the embedded system is not running any application task, it switches its state from running to idle state. The (DPM) technique keeps the system into low-power states whenever it is in idle state. This technique improves power conservation capabilities by changing selectively the multiple idle states taking into account the cost of transitions power [59, 29].
Furthermore, the prediction of the system workload could be exploited by the DPM technique to save the energy consumption by switching off or decreasing the performance of system components when they are idle or partially unexploited [102]. But, the disadvantage of using the DPM technique is that processor transitions from idle to running state requires an overhead of time and energy to serve an incoming task. Usually, the scheduler uses the DPM technique to make such decisions when executing the application based on the system state, its workload and timing constraints [38, 40, 76].
After executing an application task, the processor is able to determine the time

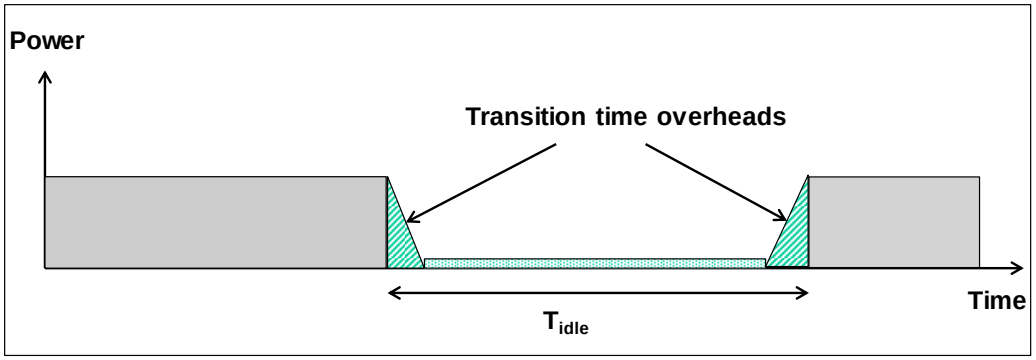


Figure 5.5: DPM technique energy saving

interval of length T_{idle} when it will be in idle state. The DPM technique compares the length of idle interval with the processor break-event time (BET), which is the minimum length of idle interval guarantying energy consumption gain when switching the processor from running to idle state [43]. The transition to idle state is performed only if T_{idle} is bigger than (BET). Figure 5.5 shows a scenario for reducing energy consumption using the DPM technique by setting the processor in idle mode, during the interval time T_{idle}, when it is not executing the application. Also, this figure shows the transition time overhead when switching from active state to idle state. To evaluate the OS services energy consumption, we use a DPM strategy named the AsDPM (Assertive Dynamic Power Management) technique proposed in [32]. This technique is based on the extraction of inherently present idleness in application's behavior to make appropriate decisions for state-transition of processors in a multiprocessor system. The AsDPM technique does not predict the time intervals when the processor is idle. This technique is based on the principle of admission control which consists in deciding when the ready task will be executed. The AsDPM low power strategy delays the execution of ready tasks as much as possible and controls the maximum number of active/running processors in the system at any time instant.

The AsDPM technique defines four types of task queue: the Tasks Queue (TQ) containing the application tasks which are neither executing nor ready at any point in time, the Released Tasks Queue (ReTQ) including tasks that are released but not running currently on any processor, the Running Tasks Queue (RuTQ) containing tasks that are released and currently running on some processors. Finally, the Deferred Tasks Queue (DeTQ) including tasks that are released but their execution is delayed. A released task, that is not the highest priority task but has its priority high enough to execute on an m-processor platform (i.e., it is among the m highest priority tasks), can be deferred from execution under AsDPM at any

scheduling event. In this technique, a runtime parameter of a task, the laxity, is used to measure task execution urgency taking into account the deadline constraint. For example, an applicative task with zero laxity is the most urgent job to execute in order to avoid deadline miss. The absolute laxity l_i of a task T_i at its release time instant t is given by equation 5.1.

$$l_i = d_i - (t + C_i) \tag{5.1}$$

where d_i and C_i are respectively the deadline and the worst-case execution time ($WCET$) of task T_i.
The working principle of AsDPM technique is showed in algorithm 3. The variable j represents the number of processors. When a scheduling event occurs, all task queues (TQ, RuTQ, ReTQ, and DeTQ) are updated and sorted according to the priority specified by the governing scheduling algorithm. Then, (j) highest priority task(s) from ReTQ are executed on (j) processor(s). For rest of the ready tasks present in ReTQ, a laxity test ($l_i \geq 0$) is performed considering the first target processor (line 6..9). If a task passes this test, it is moved into DeTQ –i.e., it is deferred from execution at current scheduling event. Otherwise, if a task does not pass this test then it implies that currently available running processors are not sufficient to satisfy the concurrent resource requirement of ready tasks and some tasks may miss their deadlines in future. In this case, all tasks which are deferred or running –i.e., present in RuTQ or DeTQ, are put into ReTQ again and more processors are activated. This procedure is repeated until ReTQ becomes empty –i.e., until all tasks present in ReTQ are either moved to RuTQ or DeTQ.

Algorithm 3 Assertive Dynamic Power Management

```
1: assign  j = 1
2: for each scheduling event do
3:   sort  TQ, ReTQ, RuTQ, and DeTQ w.r.t. scheduler's priority order
4:   repeat
5:     move  highest priority j task(s) from ReTQ to RuTQ
6:     for every remaining task i in ReTQ do
7:       if  l_i ≥ 0 on j processor(s) then
8:         move  T_i to DeTQ
9:       else
10:        move  all tasks from DeTQ and RuTQ to ReTQ
11:        assign  j = j + 1
12:        activate  j processors
13:  until ReTQ is empty
```

Modern processors support multiple power-efficient states. Since, there are temporal and energy penalties associated with state transitions, therefore, a processor

Table 5.1: Power-efficient states of OMAP3530 processor @ 125-MHz

C-state	Sleep latency (μs)	wake-up latency (μs)
Running	0	0
C1 (Idle)	73.6	78
C3 (Stand by)	163	182
C5 (Sleep)	800	366
C7 (Deep sleep)	4300	12933

needs to be put in the power-efficient state long enough to save energy. Under AsDPM, some processors of the platform have larger workload while others have less workload. For those processors having larger workload and consequently shorter idle time intervals, it is not so beneficial, some times even penalizing, to transition them into deeper power-efficient states. This is because the number of transitions on such processors is greater and accumulates large transition cost. In addition, generally, the more a state is power-efficient, the more it takes (time and energy) to recover a processor from that state. However, for other processors having longer idle time intervals, it is advantageous to put them in more power-efficient states as they are not often recovered to running state. Once the AsDPM technique has extracted idle time intervals, processors are then assigned suitable power-efficient state with respect to their worst-case workload. In our case, the OMAP 3 processor has five power-efficient states, called C-states, as shown in table 5.1, which allows dynamic power management. This table presents the sleep and wake-up latency of each C-state which are respectively the time latency from the running state to the C-state and the time overhead when switching the processor from C-state to the running state [13].

5.2.2 The DSF scheduling policy:

Nowadays the Dynamic Voltage and Frequency Scaling (DVFS) techniques have emerged. They have been particularly distinguished by their efficiency to reduce CPU power consumption. It can execute various tasks of an application at different couples of voltage/frequency depending on the workload of the processor.
The DVFS techniques adjust dynamically the voltage and frequency of the processor to minimize the energy consumption. The voltage/frequency switching mechanisms are coupled with the scheduling techniques and policies to preserve the feasibility of schedule and respect the time constraints. Several strategies have been proposed to exploit certain aspects of DVFS and offer a particular method to build pseudo intermediate frequencies for use in conjunction with the techniques of Dynamic Voltage Scaling (DVS) [82, 48].

The DVFS techniques are classified into intra-task and inter-task techniques [94, 93]. The inter-task DVFS technique based on redistribution of slack time between tasks which are ready for execution [89, 97, 116]. Consequently, the inter-task DVFS techniques make decisions related to slack reclamation only at scheduling events when ready tasks are chosen for execution. The intra-task DVFS techniques reallocate the slack time inside the same task. This kind of technique includes modules in application's code in order to study its power variation over its execution time. The intra-task voltage scaling methods have many disadvantages such as requiring excessive analysis, the feasibility of application source code update [114, 71]. Also, the intra-task DVFS technique generates an additional number of voltage and frequency switching points and most of them assume continuous voltage levels [98].
For this reason, to evaluate the energy overhead of OS services, we adapt an inter-task of DVFS technique: the Deterministic Stretch-to-Fit Technique (DSF) proposed in [31], it is based on the slowdown strategy of reducing the processor power consumption. Slowdown is known to reduce the dynamic power consumption at the cost of increased execution time for a given computation task. It detects early completion of tasks and exploits the processor resources to reduce the energy consumption.
As showed in figure 5.6, by comparing the actual execution time (AET) of a task T_1 with its worst-case execution time ($WCET$) C_1, (DSF) technique determines the value of the dynamic slack (ε). This slack time is exploited by the method to reduce the energy consumed, by stretching the execution of T_2, having C_2 as WCET, and reducing the frequency of the processor. The parameter t_{disp} is the available time at current processor frequency f. The variables t_1 and t_2 represent respectively the activation date of T_1 and T_2, d_1 and d_2 represent respectively the deadline of T_1 and T_2. Let us highlight that it is not possible to determine the exact actual execution time of the running task until it terminates, the algorithm computes the value of dynamic slack boundaries only. In addition, this slack as the difference between (WCET) and (AET) allows to reduce the speed of lower priority tasks.
The OMAP 3530 processor supports five discrete voltage and frequency levels, as shown in table 5.2 allowing static and dynamic voltage and frequency scaling.

5.3 Embedded OS services energy overhead:

In this section, we detail how to calculate the OS energy overhead when running application tasks at fixed and dynamic frequency.

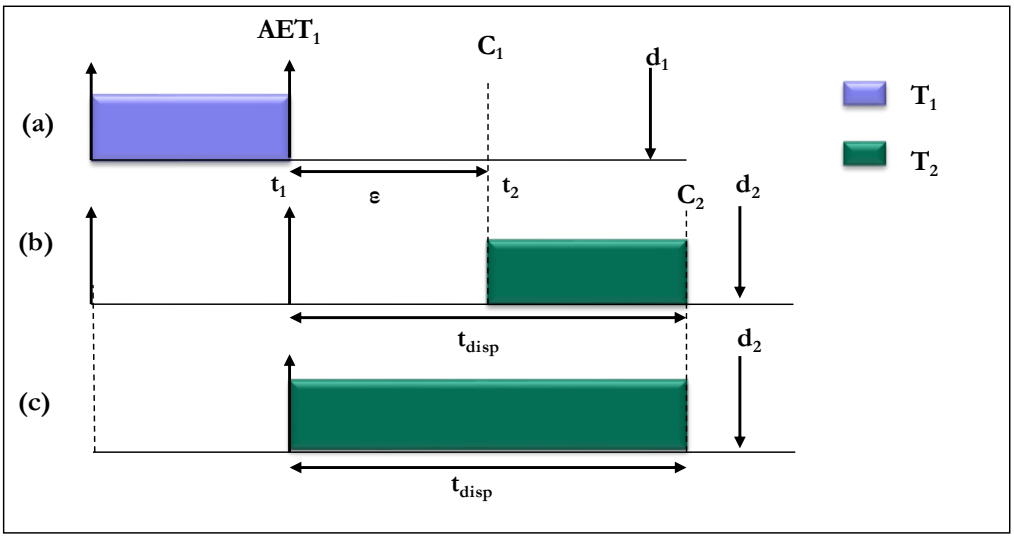


Figure 5.6: Slack reclamation using the DSF technique

5.3.1 Fixed frequency case:

When running the applicative tasks at fixed frequency, the OS services are called when switching from one task to another. To explain the calculating methodology of energy consumption when executing the software application on the hardware platform, we consider an example of task's scheduling depicted by figure 5.7. It shows the execution of three tasks and the different OS calls. Equation 5.2 shows the total energy consumed by different services of the OS.

$$E(OS) = \sum_{1 \leq k \leq n_{OS}} (E_OS_call)_k \tag{5.2}$$

where $E(OS)$, n_{OS} and $(E_OS_call)_k$ represent respectively the total energy consumption of OS services, the number of OS calls and the elementary energy consumed when the OS routines are running.

5.3.2 Dynamic frequency case:

To calculate the energy overhead of OS services when the processor changes its frequency, we consider, according to the energy analysis and modeling of OS services presented in last chapter, that OS routines are called when the processor changes the execution from one task to another one or when it varies its running frequency, especially for the context switch service. To explain the OS energy estimation, a set of two tasks T_1 and T_2 are considered. They are preempted and running on different frequencies F_1 and F_2 as depicted by figure 5.8, where:

Table 5.2: Voltage-frequency levels of OMAP330 processor

Parameter	Operating point 1	Operating point 2	Operating point 3	Operating point 4	Operating point 5
Frequency (Mhz)	125	250	500	550	720
Voltage (V)	0.975	1.05	1.2	1.27	1.35
Running power (mW)	57	130	303	348	550
Idle power (mW)	4	7	16	18	28

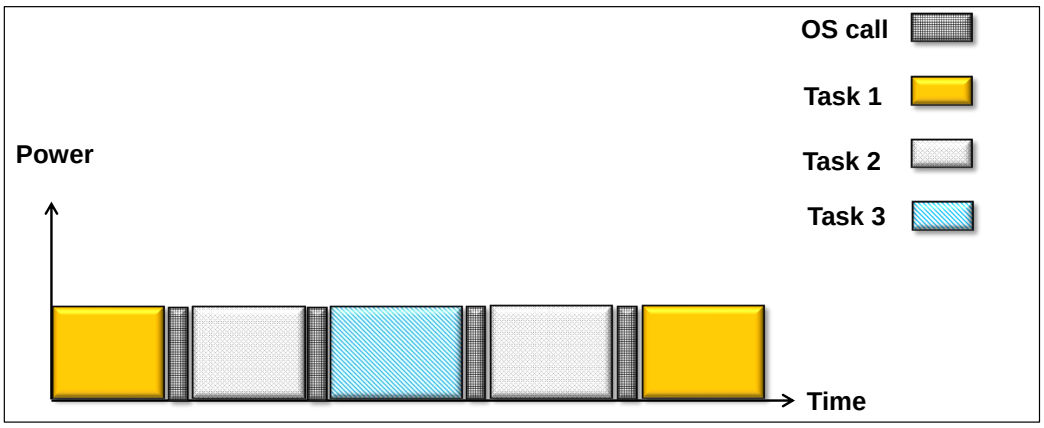


Figure 5.7: Os calls at fixed frequency

-P_{Task1} and P_{Task2} represent respectively the power consumed by $Task1$ and $Task2$.
-T_{idle} and T_{OS} represent respectively the idle time of processor and the execution time of OS routines.
-$T_{1,F1}$ and $T_{2,F1}$ represent respectively the execution time of $Task1$ and $Task2$ when the processor frequency is F_1.
-$T_{1,F2}$ and $T_{2,F2}$ represent respectively the execution time of $Task1$ and $Task2$ when the processor frequency is F_2.
Equation 5.3 details the energy consumption of OS services $E(OSV)$ when switching the processor frequency.

$$E(OSV) = \sum_{1 \leq i \leq n_{F1,F2}} (E_OS_call)_{F1,F2} + \sum_{1 \leq j \leq n_{F2,F1}} (E_OS_call)_{F2,F1} \quad (5.3)$$

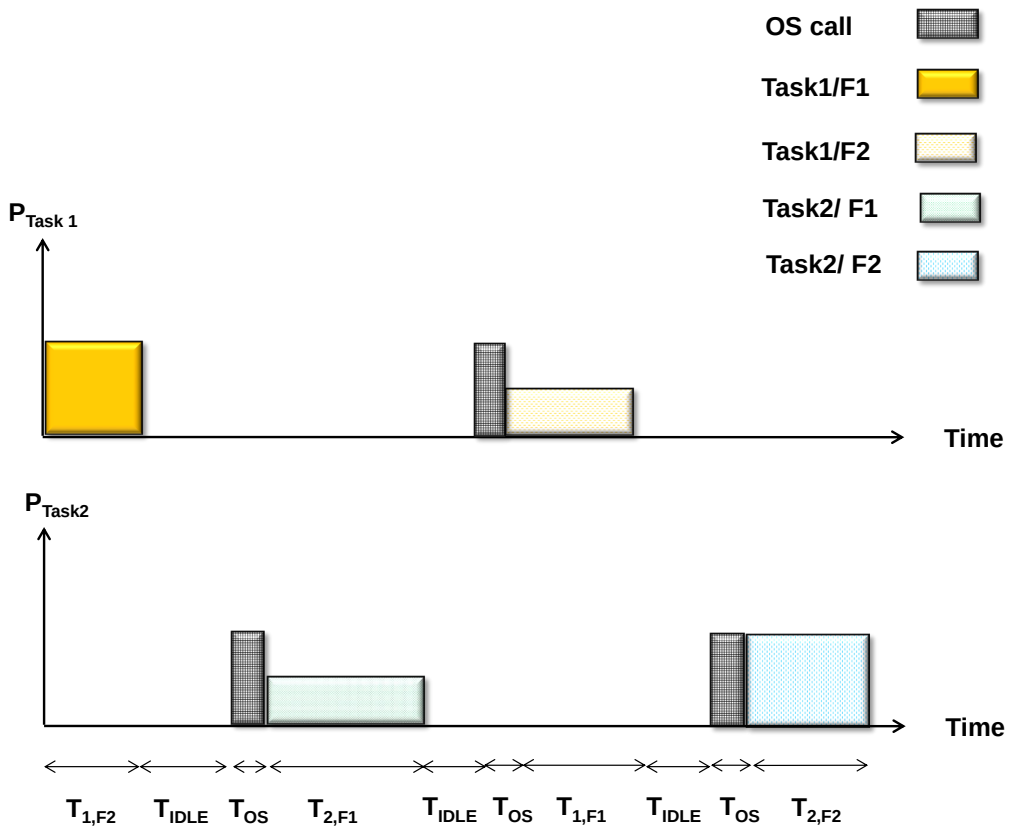


Figure 5.8: Os calls when changing the frequency

where:
-$n_{F1,F2}$ and $n_{F2,F1}$ represent respectively the number of frequency changes from F1 to F2 and from F2 to F1.
-$(E_OS_call)_{F1,F2}$ and $(E_OS_call)_{F2,F1}$ represent respectively the OS energy consumption when switching from F1 to F2 and from F2 to F1.

5.4 Experimental results:

To evaluate the OS energy overhead, the execution of the H.264 video encoder application tasks is simulated using the STORM environment. For hardware platform, we use the OMAP3530 processor to carry-out simulations. The power consumption parameters presented previously in table 5.1 and table 5.2 are used in all our simulation results. The used H.264 application provides 15 frames per second.
Software tasks are scheduled over identical multiprocessor platforms of type symmet-

ric shared-memory multiprocessor (SMP). In this architecture, two or more identical processors are connected to a single shared main memory, have full access to all I/O devices, and are controlled by a single OS instance. The processors are treated equally, with none being reserved for special purposes. Each processor executes different programs and is able to share common resources (memory, I/O device, interrupt system and so on) with other processors. These processors are connected to each other using a system bus.

Figure 5.9 shows the simulation traces of application tasks scheduling, between 1 and 50 ms, when using the DSF technique. The OS services energy consumption rates when using the DSF technique are presented in table 5.3, the initial processor(s) frequency is 500 Mhz. We note that the context switch is a basic service because the DSF technique performs many processor frequency changes in order to reduce the energy consumption. When the number of processors increases, the energy overhead of the OS services decreases because the total application execution time is reduced and the OS calls are minimized. Hence, in multiprocessor platform, it is not necessary to call the context switch service to switch from one task to another because each applicative task is running in an independent execution unit. In table 5.4, the OS services energy consumption rates using AsDPM technique are presented. This technique is less influenced by the context switch and scheduling routines energy overhead because AsDPM targets to save the energy overhead by keeping the processor in idle modes, with fewer preemptions and context switches. Also, when the processor is in idle state, these basic services consume less power than when it is in active mode.

As explained previously, the energy consumption is divided between intra-task and inter-task instructions and routines. Consequently, intra-task routines and instructions ie. application standalone tasks consume the remaining amount of energy. Figure 5.10 compares between the total energy consumption and OS services energy overhead when using DSF scheduling policy. The number of processors is 4 and, for each simulation setup, we vary the initial running frequency of each processor. When note that for initial high frequencies (500 Mhz and 720 Mhz), the energy consumption of OS services is higher. This is because studied scheduling policy reduces operating frequency to low values (125 Mhz) in order save the energy consumption which leads to significant context switch energy overhead, as detailed in chapter 3, when the difference between frequencies is high.

Under the Open PEOPLE project, modeling of power/energy of application tasks is achieved by an academic partner, INRIA of Lille, in [86, 85].

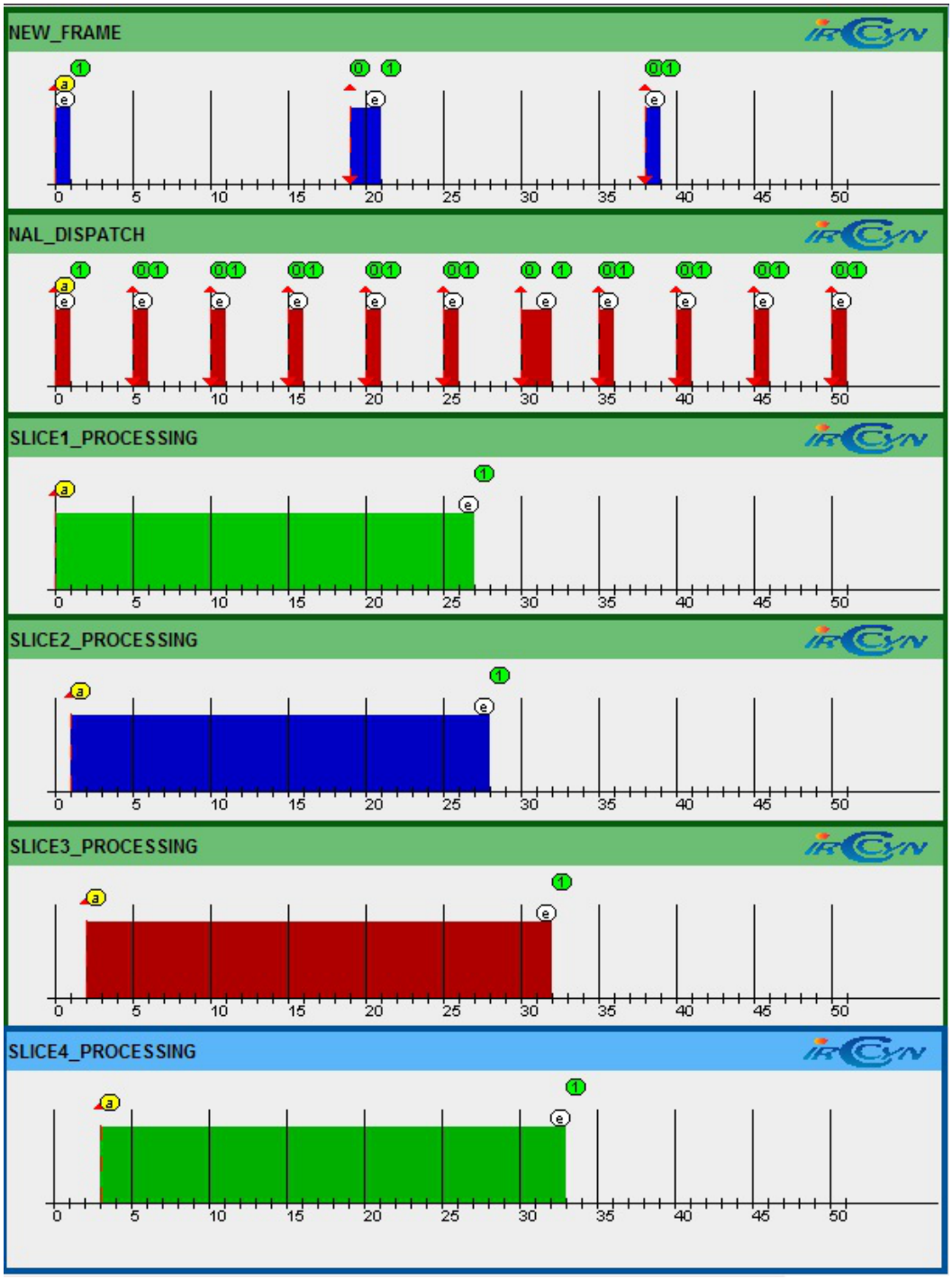


Figure 5.9: Schedule of application tasks using DSF technique

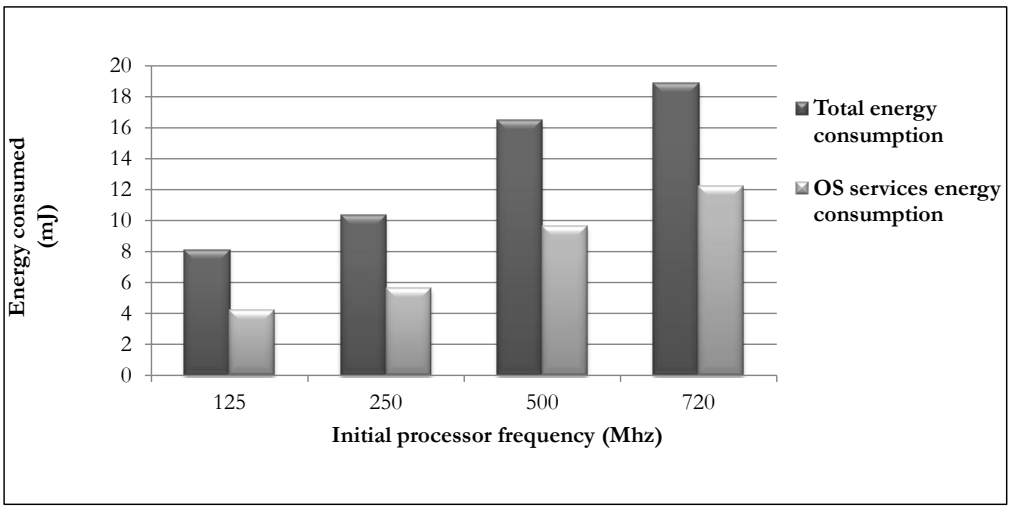


Figure 5.10: OS services and standalone application tasks energy consumption (DSF technique)

Table 5.3: OS services energy consumption rates when using the DSF technique

Number of processors	Context switch	Inter-process communication	Scheduling routines
1	44%	35.4%	1.07%
2	35.38%	32.5%	1.12%
4	30.88%	27.1%	1.38%
6	26.8%	21.7%	1.8%
8	24.62%	22.4%	1.9%

Table 5.4: OS services energy consumption rates when using the AsDPM technique

Context switch	Inter-process communication	Scheduling routines
18.67%	29.16%	1.38%

5.5 Conclusion

In this chapter, models of OS services, extracted in chapter 3, have been integrated at system level using the (STORM) simulator in order to evaluate the OS energy overhead when using AsDPM and DVFS low power techniques. Furthermore, a global approach of models integration is introduced. It is based on three focal concepts: AADL Modeling, code transformation from AADL to STORM and OS services energy and power estimation. Experimental results show that the OS ser-

vices consume a significant part of energy and that it depends on the behavior of the low power technique used.

Chapter 6

System design space exploration and verification of constraints

Contents

6.1 AADL exploration of hardware software solutions 97
6.2 Design space exploration methodology 98
6.3 System constraints definition and verification flow 99
6.3.1 RDAL Language and RDALTE tool 100
6.3.2 The Object Constraint Language (OCL) 100
6.3.3 The Quantitative Analysis Modeling Language (QAML) . . . 101
6.3.4 The proposed approach 101
6.4 System requirements analysis, definition and verification . 101
6.4.1 Quantitative analysis specifications using the QAML language 101
6.4.2 Requirements definition and verification using RDALTE tool 104
6.5 Example 109
6.6 Conclusion 111

Embedded systems often need to comply with particular requirements such as energy consumption, time constraints and processor workload. The software components binding on the hardware platform needs to take into account these platform restrictions and constraints and respect the hardware platform resources. In this chapter, we present a system design exploration methodology and we define a global flow, using a set of tools: RDALTE and QAML, to verify system requirements when allocating applicative tasks to the processors.

6.1 AADL exploration of hardware software solutions

After modeling the hardware and software components, OS services and the binding of software tasks on the hardware platform, AADL language is exploited to explore the set of possible solutions taking into account various requirements such as the processor workload.
According to hardware components specifications and software tasks characteristics,

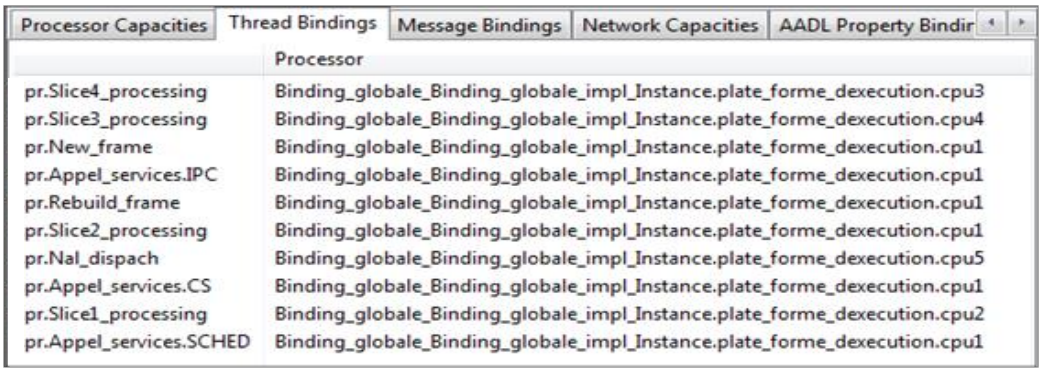


Figure 6.1: Possible solution of AADL software tasks binding on hardware platform

different solutions of software executions on the hardware platform are possible. For this reason, we exploit the AADL modeling because it allows the definition of tasks deployment on hardware components from which an analysis tool can verify the feasibility. For instance, the processor can not exceed its workload when running the software application.
The AADL exploration verifies mainly the processor workload and the instruction per cycle rate. Figure 6.1 shows a model of hardware platform which contains five processors, running at the same frequency. Then, we allow to all processors to execute the H.264 software application without deployment tasks, using the *Allowed_ Processor_ Binding* property. By choosing the partitioning strategy and fixing the running mode (operating point), we obtain an optimal task configuration (deployment) which respects the workload of each processor.
To refine this exploration, various requirements are defined. In the remaining of this chapter, a design exploration methodology is proposed. Furthermore, a flow, using a set of tools that defines and verifies system constraints, such as the OS energy consumption and scheduling requirements, is proposed.

6.2 Design space exploration methodology

As showed in figure 6.2, the exploration methodology includes three main steps. The first step of this strategy consists in searching the operating point that satisfies the maximum number of system requirements. Once the operating point is checked and validated, the design model can be reviewed and updated. The second step consists in finely reducing the exploration domain by limiting the number of execution units. The target of third and last step is the allocation of execution resources to each thread once the operating point and processor numbers of our system are

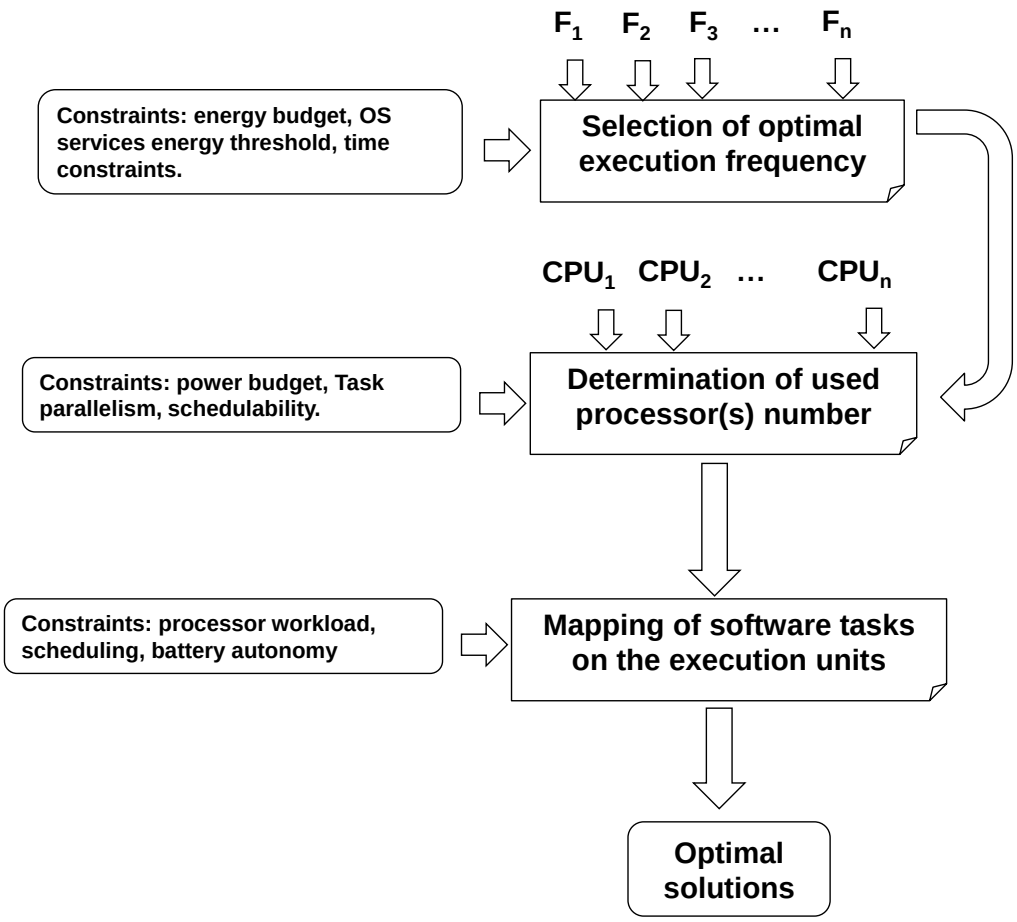


Figure 6.2: System design exploration methodology

predicted and fixed beyond the previous two levels. We note there are three classes of constraints to be satisfied depending on the exploration level. The first class includes resource consumption constraints in term of available execution resources and their energy consumption. The second constraints class concerns the power budget, parallelism and schedulability requirements. Finally, we verify operating system specific constraints.

6.3 System constraints definition and verification flow

In this section, we present different languages and tools used to specify system constraints. Then, the proposed approach of requirements definition and verification is introduced.

6.3.1 RDAL Language and RDALTE tool

To specify the system constraints, project partners choose RDAL language and RDALTE tool [21] defined below.

6.3.1.1 RDAL Language

The RDAL (Requirements Definition and Analysis Language for AADL) is a language that defines a set of requirements specifying, at different levels of details, what the system to be implemented should do. They are therefore a very important part of the system under design and are created at the very beginning of a project. When performing its functions, the system architecture takes into account the defined requirements and constraints.
In a standard requirements analysis process, an initial RDAL specification is created to model high level requirements before any system architecture is defined. The RDAL specification is then further refined into finer requirements derived from the high level requirements. In that way, the development of the requirements specifications is used to drive the design and implementation of the system. Once the requirements are sufficiently refined, the definition of a system architecture model can start. Components of the architecture can then be linked to requirements of the RDAL specifications. The requirements can be expressed in terms of a formal language (such as OCL or REAL) so that properties of the design components can be checked automatically to see if the components satisfy or meet their requirements.

6.3.1.2 RDALTE tool

The Requirements Definition and Analysis Language Tool Environment (RDALTE) has been developed by the Lab-STICC research lab in the frame of the OpenPEOPLE project. It is a set of plug-ins on top of the open source eclipse platform [46] that provides a toolset for front-end processing of RDAL specifications and for their verification over architecture models. Also, the RDALTE tool allows the creation of RDAL specifications with the help of graphical and object tree editors. Furthermore, it defines the requirements referenced model elements of the design and the expression in terms of natural or formal languages such as the OMG Object Constraint Language (OCL) or the Requirement Enforcement and Analysis Language (REAL). In this work, we use the OCL language to specify these requirements referenced model.

6.3.2 The Object Constraint Language (OCL)

The Object Constraint Language (OCL) [19] is an expression language that describes constraints on object-oriented languages and other modeling artifacts. OCL

language specifies constraints and other expressions attached to their models.

6.3.3 The Quantitative Analysis Modeling Language (QAML)

The Quantitative Analysis Modeling Language (QAML) language [20] is created during the project OPEN-PEOPLE to formally represent quantitative analysis of embedded systems models. (QEML) tool is also developed in order to exploit the QAML language and evaluate QAML specifications. Models representing quantitative analysis of all kinds can be created and associated with specific components of system architecture models. These models are then evaluated to provide estimation for properties of components to which they are associated with. While most analysis of the Open-PEOPLE project concern power and energy consumption, special care was taken during the design of the QAML language to make it generic enough so that quantitative analysis for arbitrary quantities (execution time, costs, latency, bandwidth, etc..) can be represented.

6.3.4 The proposed approach

As depicted in figure 6.3, a flow of definition and verification of system requirements is proposed. The AADL models of software and hardware components are analyzed quantitatively using the (QAML) language and the QEML tool. The definition and analysis of system requirements are performed using the (RDAL) language and RDALTE tool. The formal language OCL is used to describe different constraints and to communicate between AADL and QEML models. In fact, various OCL queries are implemented to verify different system requirements, they are collected in a constraint library in addition to the predefined OCL ones. All these libraries make writing OCL constraints much easier for the designer, when searching properties from AADL architecture and QAML quantity models.

6.4 System requirements analysis, definition and verification

In this section, we introduce the QAML quantitative analysis and we specify the composition and estimation laws used to verify the system constraints. Then, we define and verify system requirements using the RDAL tool.

6.4.1 Quantitative analysis specifications using the QAML language

A model represented with the QAML language establishes a relationship that can be evaluated to produce an output quantity from several input quantities. For example,

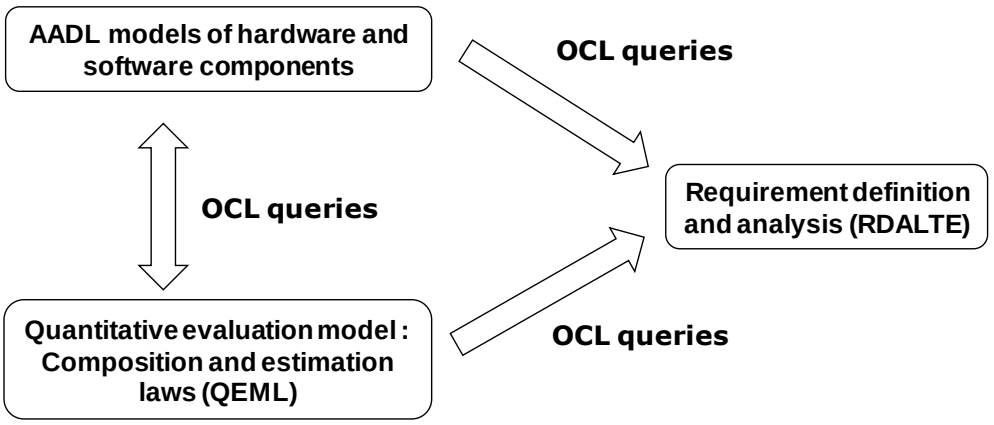


Figure 6.3: System constraints definition and verification flow

a quantity model for the power consumed by a bus may relate quantities, such as the bus frequency, the bus number of lines etc.., to the resulting power consumption quantity. There are two main types of quantity models: estimation and composition models. We propose below the implemented estimation and composition laws that will be used later to check system requirements.

6.4.1.1 QAML Estimation modeling:

The estimation model is a quantity model that also carries another estimation model for representing the uncertainty of the evaluated value. This model can be expressed in two ways: as a set of mathematical formulae (laws) or by a multi-dimension lookup table (LUT).
The estimation laws are composed of one or more pieces, which consist of a pair of formulae that define the value of the law and the domain of validity into which the corresponding first formula applies. The union of all of the pieces constitutes the estimation model itself. In this work, we associate these laws to many components of the system architecture model specifically the software components, the application threads and OS services. We define a set of estimation laws that will be used to verify system constraints:

- **Energy consumption estimation law:** This law depends on the execution time and the power consumption. For instance, the quantitative analysis view, showed in figure 6.4, represents the energy estimation law associated with an AADL system architecture component which is "$slice2_processing$" application thread, running at 720 MHz. As a result, the calculated energy

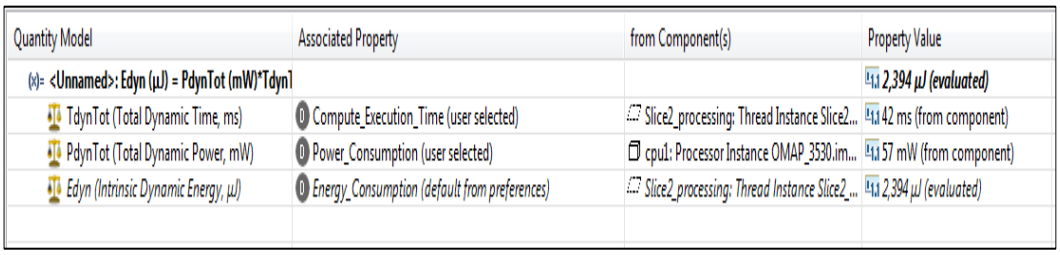


Figure 6.4: Energy consumption estimation law of *slice*2*_processing* thread running at 720 Mhz

consumption becomes an associated property to this process.

- **Execution time estimation law:** This estimation law interests on tasks execution time when running at different operating points. Knowing the execution time of each task at maximal frequency, this law calculates the execution time of different tasks at different frequencies.

- **Task CPU utilization law:** The task CPU utilization law is defined using the QAML language as the worst-case task execution time divided by its period.

6.4.1.2 QAML composition modeling:

The composition model is used to represent how quantities are combined to calculate a resulting quantity. For example, the sum of the product of power and time quantities over a set of system components contained in a system is a composition model for calculating the energy consumed by the system. As opposed to an estimation model, a composition model does not have an attached uncertainty model. However, a set of elements must be provided to which the composition function will be applied.
In the proposed model, the composition laws are used mainly to calculate the energy consumption of applicative tasks and OS services and to evaluate the CPU utilization rate.

- **Energy consumption composition law:** As showed by figure 6.5, this composition law calculates the total energy consumption of the software application by summing the elementary energy consumption of all application tasks and OS services. This total energy consumption composition law is sub-divided under two composition laws. The first one calculates the energy consumption of different AADL software threads. For this, OCL queries are used to collect different threads allowed to be executed on platform processors. The second law determines the energy overhead of OS services: it multiplies

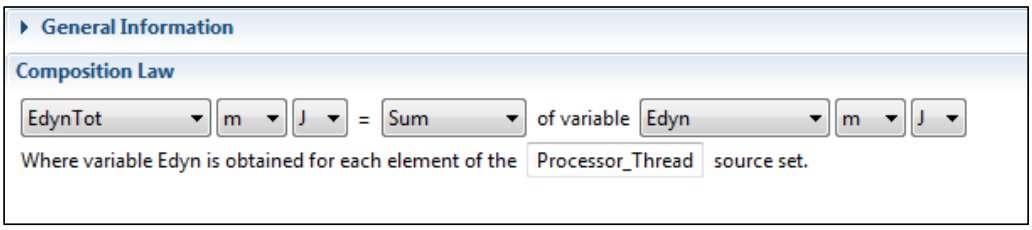


Figure 6.5: Energy consumption composition law

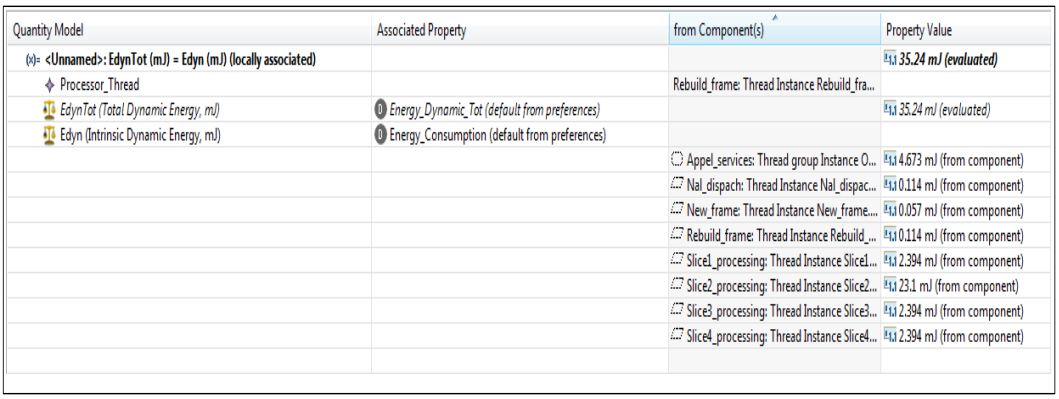


Figure 6.6: Evaluation of energy consumption composition law

the energy consumed by each OS call by the number of OS calls. The evaluation of the energy consumption composition law applied to software tasks and OS services is showed in figure 6.6.

- **Tasks CPU utilization composition law:** This law calculates the sum of different application and OS threads CPU utilization rates. The obtained value will be used later to verify schedulability constraints.

6.4.2 Requirements definition and verification using RDALTE tool

In this section, we detail the different constraints that the studied embedded system should satisfy or that we have to verify in the design exploration process. Some constraints concern software application components; some others are formalized to check the efficiency of hardware components. Also, we implement some requirements that revolve around the integration of OS services in the AADL model.
The schematic diagram, represented in figure 6.7, shows a hierarchical view of an RDAL model and defines non-functional requirements that should be verified according to their priorities. As explained previously, three classes of constraints are defined depending on the exploration level:

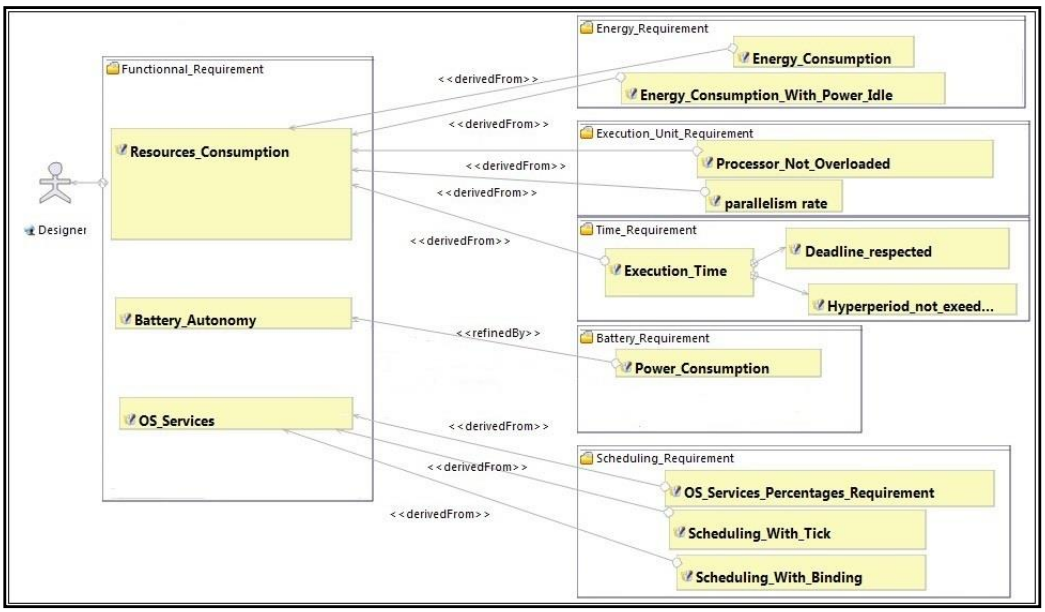


Figure 6.7: System requirements RDAL diagram

6.4.2.1 First level of design exploration constraints:

The set of constraints verified at this level are:

- **Energy consumption Requirement:** The energy consumption requirement is attached to the process instance that contains the different threads of the H.264 application, including the OS services as specific threads. For this constraint, an energy budget is fixed. The total energy consumption should respect allowed budgets for used system and should not be exceeded when running the application on the hardware platform. This constraint also allows checking different possible configurations. Thus, we can ensure for each operating point an efficient exploitation of the energy resource. The chosen value helps us to specify which configuration can be a candidate for the next exploration level. The global energy consumption will be evaluated as in the equation below 6.1

$$\sum_{1 \leq i \leq n} E_{T_i} \leq EB \tag{6.1}$$

  Where n, E_{T_i} and EB represent respectively the number of application threads, the energy consumed by the thread number (i) and the energy budget. To verify this constraint, the QAML energy consumption estimation law and its associated property to each process, described in the section above, are used. Once getting the energy consumption of each thread at the running frequency, the OCL queries will browse the tree view of AAXL model and

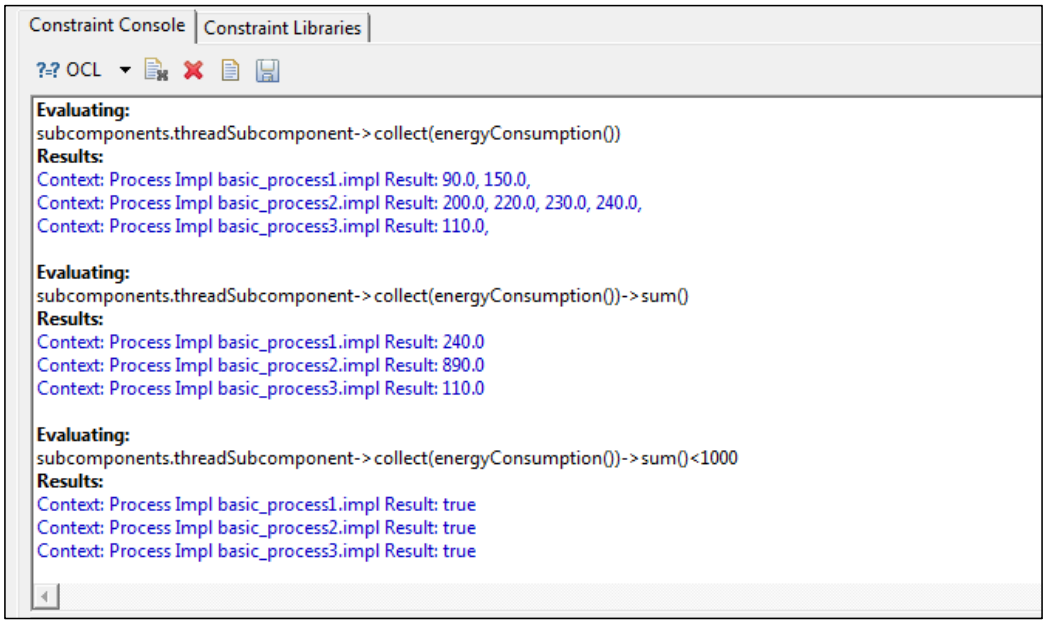


Figure 6.8: Energy consumption requirement verification using OCL

accumulate threads energy consumption, contained in the process under test. The verification of this constraint is done by returning a boolean value and the requirements verification rates are displayed in the AADL project navigator view. An overview of OCL editor is shown in figure 6.8. It details the total energy consumed and the constraint evaluation.

- **OS services energy consumption requirement:** This constraint focuses on the energy consumption of the studied services: scheduling, context switch and the inter process communication services. This requirement verifies whether the OS service energy consumption percentage does not exceed a percentage of the global energy consumption. This percentage is fixed by the designer. The expression "$Edyn_Tot$", evaluated with QEML estimation and composition laws, presents the total energy consumed by threads and OS services. We can then deduce services energy percentages thanks to OCL queries. Knowing the number of OS calls when processing one frame, this requirement will be satisfied for each OS service.

- **Timing requirements:** The execution time constraints remain critical in embedded system modeling. In this study, two types of timing requirements are verified: execution time and deadline requirements.
  -Execution time requirement: Depending on the hardware platform, the

total application execution time should not exceed a threshold time. The total application execution time is computed by summing the contribution of tasks, sequentially executed, and the maximum of case execution time of threads which are executed in parallel. The analytic expression 6.2 should be respected to satisfy this constraint.

$$\sum_{1 \leq i \leq ns} CET_{Ts_i} + max\{CET_{Tp_j}, 1 \leq j \leq np\} \leq TT \tag{6.2}$$

Where ns and np represent respectively the number of serial and parallel tasks. The set of serial and parallel tasks are respectively $\{\{Ts_i\}, 1 \leq i \leq ns\}$ and $\{\{Tp_j\}, 1 \leq j \leq np\}$, their execution times are CET_{Ts_i} and CET_{Tp_j}. The execution time threshold is TT.

-Deadline requirement: This requirement specifies, for each task, that the case execution time property, once evaluated at the specific operating point, must met the corresponding deadline. For a specific operating point, the execution time for each thread is deduced using the rule of three and the quantity model. Otherwise, we consider that execution time function is linear as showed in equation 6.3.

$$CET_{F_i} = CET_{F_{max}} \times F_{max}/F_i \tag{6.3}$$

Where F_i and F_{max} are respectively the running and maximal frequency. CET_{F_i} and $CET_{F_{max}}$ represent respectively the execution times at frequency F_i and F_{max}.

6.4.2.2 Second level of design exploration constraints

At the second level of design exploration, we have to take into account these requirements:

- **Power budget Requirement:** This requirement concerns battery powered systems. It verifies that each processor's power consumption does not exceed a fixed value depending on the battery autonomy. In order to evaluate the global power consumed by all hardware platform components, we need to fix the processor number. This expression 6.4 is used to calculate total power consumption.

$$P = K_{CPU} \times V^2 \times F \times N_{CPU} \tag{6.4}$$

Where K_{CPU} represents a specific constant for processor; in our case, we used an OMAP 3530 processor and we recall that we adopted the homogeneous embedded systems. (F,V) is the used operating point: the running frequency and voltage. N_{CPU} represents the processor number used in the platform.

```
context PropertyHolder def : is_Schedulable():

Boolean =
          averagetogglerate()<(instance::ComponentInstance.allInstances()->select(isProcessor())->size()+1)/2
```

Figure 6.9: OCL expression of the schedulability test

The global power must be less than the autonomy of the battery chosen by the designer.

- **Parallelism requirement:** When running the software application, the parallelism constraint should be respected. For that, the number of processors should be greater than or equal to the parallelism rate which is the number of slices that will be executed concurrently. We remind in this context that a processor can execute only one task at once a time.

- **Global scheduling requirement:** This schedulability requirement specifies the least processor number that hardware platform can support to execute the application tasks and verify their deadlines. Since we are interested in global scheduling, this constraint is checked before scheduling tasks on hardware execution units. This specificity is due to the homogeneity of used hardware platform. We start by checking one of the most used scheduling policies in the multiprocessor embedded systems: Earliest Deadline First (EDF) [36, 67]. Thus, the schedulability test of EDF is presented in expression 6.5:

$$\sum_{1 \leq i \leq n} CET_i / P_i \leq (m+1)/2 \tag{6.5}$$

Where m is the processor number in the multiprocessor platform and n the number of tasks. The first part of the inequality is the sum of different tasks CPU utilizations. Application tasks, satisfying this requirement, are schedulable by m-processor platform. This condition is necessary and sufficient to confirm the schedulability test. The computed task CPU utilization, which is the result of the QAML estimation law, and the global application workload, which is the sum of the different task CPU utilizations, are computed using QAML composition laws. The OCL expression checking this constraint and validating the schedulability test, is showed in figure 6.9.

6.4.2.3 Third level of design exploration constraints

Depending on the running mode presented in AADL deployment model, the binding of each task to the suitable execution target (CPU) is performed. At the third level of design exploration, AADL binding properties are used to identify which

are software components that handle specific hardware components. For this, it is recommended to use the *Allowed_ Processor_ Binding* in the deployment model to enable the AADL scheduler to affect the different application tasks to the appropriate execution target taking into account their deadlines as well as the parallelism rate. We have to add specific requirements in this level in addition to the energy and timing constraints defined in previous step. We will detail in this section how to implement the schedulability and the workload requirements for each processor.

- **Processor workload requirement:** This requirement is applied to the instantiate processor set, independently to tasks deployment on the hardware platform; the verification of this requirement warns us of the processor overload cases. An OCL query, implemented to identify for each processor its allocated tasks, will compare the sum of these tasks CPU utilizations (the case execution time divided by its period) with 100%.

- **Scheduling requirement:** When allocating application tasks for processors, the schedulability condition for each processor following the standards scheduling policy: Earliest deadline first (EDF), Rate Monotonic (RM) is checked. Expressions 6.6 and 6.7 are verified respectively to confirm the scheduling requirements of (EDF) and (RM) policies.

$$U_i \leq n \times (2^{1/n} - 1) \tag{6.6}$$

Where U_i represents the workload of the processor number (i) and n its allowed tasks number.

$$\sum_{1 \leq i \leq n} CET_{T_i}/P_{T_i} \leq 100 \tag{6.7}$$

Where CET_{T_i} and P_{T_i} are respectively the case execution time and the period of task T_i.

6.5 Example

An example of verification scenario is used in order to explore the design and evaluate the different constraints. When executing the H.264 application presented previously in chapter 4, the total energy consumption, the execution time with their deadline satisfaction percentages and finally the OS energy overhead are measured. Table 6.1 shows the total energy consumption, the OS energy rates and allowed execution times of different possible solutions. The OS energy overhead should not exceed 60% of the total energy consumption.
As explained in the first level of our strategy, we can eliminate the highest frequency (720 Mhz) in view of exceeding the energy budget fixed by the designer. We note also

Table 6.1: Characteristics of possible solutions

Frequency (Mhz)	OS energy consumption rate (%)	Total energy consumption (mJ)	Allowed time (ms)
125	35%	57.4	278
250	36.38%	65.3	139
500	52.5%	80.1	74
720	62.8%	98.7	48

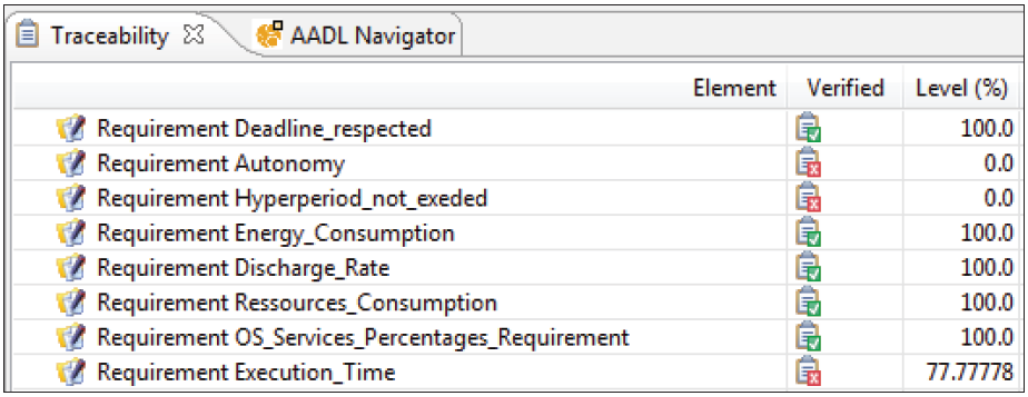


Figure 6.10: Requirements satisfaction rates (500 Mhz)

that the least operating point (125 Mhz) does not respect both of tasks deadline and allowed execution time. The requirements satisfaction rates when running on the operating point (500 MHz) are presented in figure 6.10. Because of its requirements satisfaction rate, this frequency is chosen in the first step of design exploration.
In the second step, after fixing the running frequency, the number of processors should be selected. This number should verify the maximum of second level of design exploration constraints. In our case, the number of processors varies from 4 to 8. The selected processors number, having the highest satisfaction rate of the different requirements checked in the second level of design exploration (72%), is five. Finally, we perform the tasks partitioning using the AADL tools. The used scheduling policy is Rate Monotonic. The schedulability test for each processor when AADL decides tasks partitioning is checked. As showed in figure 6.11, the average acceptance rate of third level of exploration constraints is (90%).

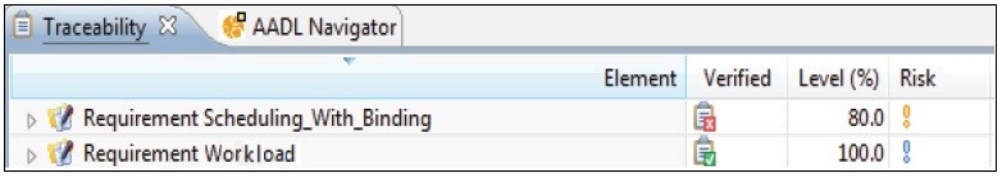


Figure 6.11: Satisfaction rates of third level of exploration requirements

6.6 Conclusion

In this chapter, we have presented various requirements in order to refine the design exploration. Depending on the exploration level, three classes of constraints are satisfied. The first class checks resource consumption in term of available execution resources and their energy consumption. The second constraints class concerns autonomy requirements and the last class verifies operating system energy consumption and scheduling policies constraints. Also, a flow, using a set of tools that defines and verifies system constraints, is proposed. The AADL models of software and hardware components are analyzed quantitatively using the (QAML) language. The definition and analysis of system requirements are performed using the (RDAL) language and RDALTE tool. Also, the formal language (OCL) is exploited to describe different constraints and to communicate between AADL and QEML models.

Chapter 7

Conclusion

Contents

7.1 Conclusion 113
7.2 Perspectives 114
7.2.1 OS services energy characterization approach extension . . . 115
7.2.2 OS services energy optimization 115
7.2.3 System level thermal modeling 115

This chapter summarizes the proposed methodology of characterization, estimation and modeling of OS services energy consumption and recapitulates the main thesis contributions that have been discussed in previous chapters. Also, it presents the perspectives and future works.

7.1 Conclusion

All along this document, efforts have been made to present methodology targeting to characterize, estimate and model the power and energy consumption of embedded operating systems running on a hardware platform.
As detailed in the introduction, power consumption is a major challenge for embedded systems designers. Besides, embedded and real time OS are more and more used by embedded systems developers: nearly 73% of embedded projects integrate an OS. Various studies and research works have pointed out the overhead of some specific OS services. In this thesis, a methodology is proposed in order to characterize and model the power overhead of OS services. The proposed approach comprised of four main contributions: characterizing the power and energy consumption of OS services, the AADL modeling of different features of the used embedded system components and the OS services, the integration of OS services models in the system level design flow using low power scheduling policies and verification of system and OS services energy consumption constraints.
In chapter 3, we introduce a flow of embedded OS services power/energy consumption characterization. Furthermore, we present the methods and benchmarks used to determine energy and power overheads of a set of three basic services of the embedded OS: scheduling, context switch and inter-process communication. In addition,

we study the variation of power/energy consumption of the embedded OS services and we analyzed the impacts of hardware and software parameters like processor frequency and scheduling policy on energy consumption. An accurate mathematical models and laws of the power and energy consumption are extracted. The use-case embedded system used is the OMAP3530 EVM board with an OMAP3 processor and Linux 2.6.32 operating system. Then, in chapter 4 , AADL language is used to model OS services, applicative tasks, hardware platform and the binding of software tasks on the hardware components. The H.264 video decoder application is taken as main use case application.
Furthermore, using the power/energy models and laws of the OS basic services extracted in chapter 5, the energy overhead of the scheduling, the context switch and inter-process communication routines is determined when adapting low power techniques: the DPM and DVFS techniques. To calculate the energy and power overhead of the embedded OS services, extracted models of OS services are integrated in multiprocessor scheduling estimation tool: STORM. A global approach is introduced, it is based on three focal concepts: AADL Modeling, code transformation from AADL to STORM and OS services energy and power estimation. Taking into account the properties of the application tasks and the hardware platform, the energy overhead of OS services is calculated. Experimental results show that OS services consume a significant part of energy and that it depends on the low power scheduling policy used.
In chapter 6, we introduce a flow of definition and verification of system requirements. The AADL models of software/hardware components are analyzed quantitatively using the Quantitative Analysis Modeling Language (QAML). Also, we define a set of system requirements, such as the OS services energy consumption and scheduling Requirements, using the (RDAL) language and RDALTE tool. The formal language OCL (Object Constraint Language) is used to describe different constraints and to communicate between AADL and QAML models. Taking into account these requirements, an exploration of possible binding solutions is performed: we search the operating points and the number of execution units that satisfies the maximum number of system requirements. Then, we allocate execution resources to each thread.

7.2 Perspectives

In this section, we discuss many extensions of the proposed work and we present future works.

7.2.1 OS services energy characterization approach extension

We have studied in chapter 3 the energy consumption of three basic embedded OS services: the context switch, the scheduling and the interprocess communication. As a continuation of the work, we are planning to calculate the energy overhead of other OS services. Also, we will validate the energy characterization approach using various hardware platforms and peripherals and compare the OS energy overhead when using different architectures. Furthermore, we can extend this study by extracting power and energy models for other embedded and real time operating systems.

7.2.2 OS services energy optimization

One of the possible extension of this work can be the optimization OS energy consumption. In fact, we can define, for other complex architectures, a multi-objective function that characterizes the energy consumption of the embedded operating system. This function depends on many hardware and software parameters. The goal is to find a hardware/software binding solution that minimizes the OS energy overhead. Depending on the complexity of the architecture, we can use, to search the OS energy optimal or good solution, heuristic or complete methods. Heuristic algorithms, such as the bees algorithm, are used when the set of possible solutions is huge. The complete methods explore all possible solutions and ensures an optimal solution as a result.

7.2.3 System level thermal modeling

Future works also include the system level thermal modeling. We can develop a module aiming to characterize the temperature of processor blocks using infrared measurement framework, such as infrared cameras with high spatial resolution, that permits the capture of run-time power consumption and thermal characteristics of modern chips.

Bibliography

[1] BSIM3v3 manual. http://www-device.eecs.berkeley.edu/bsim/?page=BSIM3.

[2] Mentor Graphics Eldo simulator. http://www.mentor.com/products/ic_nanometer_design/analog-mixed-signal-verification/eldo/.

[3] SmartBadge 4 Manual. http://www.it.kth.se/∼maguire/badge4.html.

[4] Systems Modeling Language open source specification project, July 2006. http://www.sysml.org.

[5] ANR project Pherma, 2007-2010. http://pherma.irccyn.ec-nantes.fr.

[6] NVIDIA CUDA C SDK Code Samples, NVIDIA corporation, 2011. http://docs.nvidia.com/cuda/cuda-samples/index.html.

[7] OMAP35x Evaluation Module (EVM), 2011. http://focus.ti.com/docs/toolsw/folders/print/tmdsevm3530.html.

[8] PTX: Parallel Thread Execution ISA, NVIDIA, 2011. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[9] STORM simulation tool, 2011. http://storm.rts-software.org.

[10] Thales group (France), 2011. http://www.thalesgroup.com.

[11] ESTEREL Technologies: esterel studio, 2012. http://www.estereltechnologies.com/products/esterel-studio/.

[12] Open-PEOPLE project: Open power and energy optimization platform and estimator, 2012.

[13] Power management device latencies measurement, 2012. http://www.omappedia.org/wiki/Power_Management_Device_Latencies_Measurement/.

[14] Texas instruments company, 2012. http://www.ti.com/.

[15] VHDL Analysis and Standardization Group (VASG), 2012. http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome.

[16] API JDOM, 2013. http://www.jdom.org/.

[17] Object Management Group, 2013. http://www.omg.org/.

[18] Object Management Group, Unified Modeling Language: Superstructure, 2013. http://www.uml.org/.

[19] OCL language, 2013. http://www.omg.org/spec/OCL/.

[20] QAML language, 2013. http://avalon.aut.bme.hu/mpm12/presentations/pres14.pdf.

[21] RDAL language, 2013. http://cit.tu.edu.sa/web/ccs/publications/111231094019.pdf.

[22] Andrea Acquaviva, Luca Benini, and Bruno Riccó. Energy characterization of embedded real-time operating systems. *ACM SIGARCH Computer Architecture News*, 29:13–18, 2001.

[23] Sumit Ahuja, Deepak Mathaikutty, and Sandeep K. Shukla. Applying verification collaterals for accurate power estimation. In *9th International workshop on Microprocessor test and Verification (MTV)*, pages 61–66, Austin, Texas, USA, 2008.

[24] Sumit Ahuja, Deepak A. Mathaikutty, Gaurav Singh, Joe Stetzer, Sandeep K. Shukla, and Ajit Dingankar. Power estimation methodology for a high-level synthesis framework. In *Proceedings of the 10th International Symposium on Quality of Electronic Design*, ISQED '09, Santa Clara, California, USA, 2009.

[25] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[26] Algirdas Avizienis, Jean claude Laprie, and Brian Randell. Fundamental concepts of dependability, 2001.

[27] Robert Basmadjian, Florian Niedermeier, and Hermann De Meer. Modelling and analysing the power consumption of idle servers. In *Proc. of the 2nd IFIP Conf. on Sustainable Internet and ICT for Sustainability (SustainIT 2012)*, Pisa, Italy, 2012. IFIP. The original publication is available at dl.ifip.org (to appear).

[28] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce L. Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers*, 52:1454–1469.

[29] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI systems*, 8(3):299–316, 2000.

[30] Simona Bernardi, José Merseguer, and Dorina C. Petriu. Adding dependability analysis capabilities to the MARTE profile. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 736–750, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Muhammad Khurram Bhatti, Cécile Belleudy, and Michel Auguin. An inter-task real time DVFS scheme for multiprocessor embedded systems. In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 136–143, Edinburgh, Scotland, 2010.

[32] Muhammad Khurram Bhatti, Muhammad Farooq, Cécile Belleudy, Michel Auguin, and Ons Mbarek. Assertive dynamic power management (AsDPM) strategy for globally scheduled rt multiprocessor systems. In *Proceedings of the 19th international conference on Integrated Circuit and System Design: power and Timing Modeling, Optimization and Simulation*, PATMOS'09, pages 116–126, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] Andrea Bona, Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, Vittorio Zaccaria, and Roberto Zafalon. Reducing the complexity of instruction-level power models for vliw processors. *Design automation for embedded systems*, 10(1):49–67, 2005.

[34] Keith A. Bowman, Blanca L. Austin, John C. Eble, Xinghai Tang, and James D. Meindl. A physical alpha-power law mosfet model. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 218–222, New York, NY, USA, 1999. ACM.

[35] Carlo Brandolese and William Fornaciari. Measurement, analysis and modeling of rtos system calls timing. In *Euromicro Symposium on Digital Systems Design*, pages 618–625, 2008.

[36] Alan Burns and Andy Wellings. *Real-Time Systems and Programing Languages: Ada 95, Real Time Java and Real Time Posix.* Addison Wesley, 2001.

[37] George Z. N. Cai and Chee How Lim. Microarchitectural power analysis for cpu power/performance optimization. Technical report, Intel company, 2001. The SimpleScalar-Arm Power Modeling Project.

[38] Le Cai and Yung-Hsiang Lu. Dynamic power management using data buffers. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10526–, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Ozgur Celebican, Tajana Simunic Rosing, and Vincent J. Mooney, III. Energy estimation of peripheral devices in embedded systems. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, GLSVLSI '04, pages 430–435, New York, NY, USA, 2004. ACM.

[40] Eui-Young Chung, Luca Benini, Alessandro Bogliolo, Yung-Hsiang Lu, and Giovanni De Micheli. Dynamic power management for nonstationary service requests. *IEEE Transactions on Computers*, 51(11):1345–1361, 2002.

[41] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[42] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002.

[43] Vinay Devadas and Hakan Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 99–108, New York, NY, USA, 2008. ACM.

[44] S. Dhouib, E. Senn, J.P. Diguet, and J. Laurent. Modelling and estimating the energy consumption of embedded applications and operating systems. In *Proceedings of the 12th International Symposium on Integrated Circuits, ISIC'09*, pages 457–461, Singapore, 2009.

[45] Robert P. Dick, Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha. Power analysis of embedded operating systems. In *Design Automation Conference*, Los Angeles, California, USA, 2000.

[46] Eclipse Foundation. The Eclipse Project. http://www.eclipse.org/.

[47] R. Egawa, M. Ito, N. Hasegawa, and T. Nakamura. Temperature Gradient Alleviating Method for Arithmetic Units. In *International Worshop on thermal investigations of ICs and Systems*, pages 151–156, Belgirate, Lago Maggiore, Italie, September 2005. TIMA Editions.

[48] Stijn Eyerman and Lieven Eeckhout. Fine-grained DVFS using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1:1–1:24, February 2011.

[49] Farzan Fallah and Massoud Pedram. Standby and active leakage current control and minimization in cmos vlsi circuits. *IEICE Transactions*, 88-C(4):509–519, 2005.

[50] K. M. Fant and S. A. Brandt. Null convention logic/sup TM/: A complete and consistent logic for asynchronous digital circuit synthesis. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, ASAP '96, pages 261–, Washington, DC, USA, 1996. IEEE Computer Society.

[51] Y. Fei, S. Ravi, A. Raghunathan, and N.K. Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Transactions on Embedded Computing Systems (TECS)*, 7:1–26, 2007.

[52] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[53] Nicolas Fournel, Antoine Fraboulet, and Paul Feautrier. eSimu: a fast and accurate energy consumption simulator for real embedded system. In *International Symposium on a World of Wireless, Mobile and Multimedia Networks (WOWMOM'07)*, pages 1–6, Helsinki, Finland, 2007.

[54] Pat Gelsinger. Moore's law - the genius lives on. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):18–20, 2006.

[55] Bing Guo, Dianhui Wang, Yan Shen, and Zhishu Li. A hopfield neural network approach for power optimization of real-time operating systems. *Neural Computing and Applications*, 17:11–17, 2008.

[56] Raimo Haukilahti. Energy characterization of a RTOS hardware accelerator for SoCs. In *In Proc. of the Swedish System-on-Chip Conference (SSoCC)*, Falkenberg, Sweden, March 2002.

[57] Babak Hidaji, Mohamad Reza Andalibizadeh, and Salar Alipour. Micro-architectural power estimation and optimization. In *IEEE International Conference on Electro/Information Technology*, Windsor, Ontario, Canada, 2009.

[58] Jérôme Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. In *IEEE International Workshop on Rapid System Prototyping'07*, pages 106–112, 2007.

[59] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems*, 2(3):325–346, August 2003.

[60] Nasser Jazdi. Component-based and distributed web application for embedded systems. In *International Conference on Intelligent Agents, Web Technology and Internet Commerce, IAWTIC'2001*, Las Vegas, USA, 2001.

[61] Kyungtae Kang, Kyung-Joon Park, and Hongseok Kim. Functional-level energy characterization of μC/OS-II and cache locking for energy saving. *Bell Labs Technical Journal*, 17(1):219–227, 2012.

[62] Nam Sung Kim, Taeho Kgil, Valeria Bertacco, Todd M. Austin, and Trevor N. Mudge. Microarchitectural power modeling techniques for deep sub-micron microprocessors. In Rajiv V. Joshi, Kiyoung Choi, Vivek Tiwari, and Kaushik Roy, editors, *Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004, Newport Beach, California, USA, August 9-11, 2004*, pages 212–217. ACM, 2004.

[63] Brian Klug. Two OMAP 3430 phones: Nokia N900 and Motorola Droid. *AnandTech computer hardware magazine*, 2010.

[64] Vasilios Konstantakos, Alexander Chatzigeorgiou, Spiridon Nikolaidis, and Theodore Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on instrumentation and measurement*, 57(4):797–804, 2008.

[65] Johann Laurent, Nathalie Julien, Eric Senn, and Eric Martin. Functional level power analysis: An efficient approach for modeling the power consumption of complex processors. In *Design, Automation and Test in Europe Conference and Exposition (DATE 2004)*, pages 666–667, 2004.

[66] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. In *In Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS*, San Diego, California, USA, 2003.

[67] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[68] Fang Liu, Fei Guo, Yan Solihin, Seongbeom Kim, and Abdulaziz Eker. Characterizing and modeling the behavior of context switch misses. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *PACT*, pages 91–101, Toronto, CANADA, 2008.

[69] Daniel Lohmann, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *Proceedings of the 10th IEEE International Workshop on Object-*

Oriented Real-Time Dependable Systems, WORDS '05, pages 413–420, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[71] Daniel Mosse, Hakan Aydin, Bruce Childers, and Rami Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *In Workshop on Compilers and Operating Systems for Low Power*, 2000.

[72] Richard Nass. An insider's view of the 2008 embedded market study. *Embedded Systems Design*, 2008.

[73] Nathan Wayne Fisher. *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, University of North-Carolina, Chapel Hill, 2007.

[74] David Nellans, Rajeev Balasubramonian, and Erik Brunvand. Interference aware cache designs for operating system execution. Technical Report UUCS-09-002, University of Utah, February 2009.

[75] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, 2005.

[76] Gethin Norman, David Parker, Marta Kwiatkowska, Eep Shukla, and Rajesh Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17:202–215, 2003.

[77] Object Management Group. The SAE AADL Standard Info Site, 2009. http://www.aadl.info/.

[78] OMG. UML profile for schedulability, performance, and time specification v1.1, 2005.

[79] Bassem Ouni, Cécile Belleudy, Sebastien Bilavarn, and Eric Senn. Embedded operating systems energy overhead. In *Conference on Design and Architectures for Signal and Image Processing, DASIP*, pages 52–57, Tampere, Finland, 2011.

[80] Jaehyun Park, Donghwa Shin, Naehyuck Chang, and Massoud Pedram. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design (ISLPED'10)*, pages 419–424, Austin, TX, USA, 2010.

[81] Sandro Penolazzi, Ingo Sander, and Ahmed Hemani. Predicting energy and performance overhead of real-time operating systems. In *Design, Automation and Test in Europe, DATE 2010*, pages 15–20, Dresden, Germany, March 2010.

[82] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001.

[83] Nantes Real time systems group, IRCCyN research laboratory, 2009. http://www.irccyn.ec-nantes.fr/.

[84] Siddharth Rele, Santosh Pande, Soner Önder, and Rajiv Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 261–275, London, UK, UK, 2002. Springer-Verlag.

[85] Santhosh Kumar Rethinagiri, Rabie Ben Atitallah, and J Dekeyser. A system level power consumption estimation for mpsoc. In *System on Chip (SoC), 2011 International Symposium on*, pages 56–61. IEEE, 2011.

[86] Santhosh Kumar Rethinagiri, Rabie Ben Atitallah, Jean-Luc Dekeyser, Eric Senn, and Smail Niar. An efficient power estimation methodology for complex risc processor-based platforms. In *Proceedings of the great lakes symposium on VLSI*, GLSVLSI '12, pages 239–244, New York, NY, USA, 2012. ACM.

[87] Jose L. Rosselló, Carol de Benito, and Jaume Segura. A compact gate-level energy and delay model of dynamic cmos gates. *IEEE Transactions on circuits and systems-II: EXPRESS BRIEFS*, 52(10):685– 689, 2005.

[88] Stéphane Rubini, Frank Singhoff, and Jérôme Hugues. Modeling and verification of memory architectures with aadl and real. In *Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '11, pages 338–343, Washington, DC, USA, 2011. IEEE Computer Society.

[89] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. An intra-task dvfs technique based on statistical analysis of hardware events. In *Proceedings of the 4th international conference on Computing frontiers*, CF '07, pages 123–130, New York, NY, USA, 2007. ACM.

[90] M. Schneider, H. Blume, and T. G. Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2:215–219, 2004.

[91] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. Softexplorer: Estimation, characterization, and optimization of the power and energy consumption at the algorithmic level. In *14th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2004)*, pages 342–351, 2004.

[92] Eric Senn, Johann Laurent, Nathalie Julien, and Eric Martin. Softexplorer: Estimating and optimizing the power and energy consumption of a C program for DSP applications. *EURASIP Journal on Advances in Signal Processing*, 2005(16):2641–2654, 2005.

[93] Hyungjung Seo, Jaewon Seo, and Taewhan Kim. Algorithms for combined inter- and intra-task dynamic voltage scaling. *The computer journal*, 55(11):1367–1382, 2012.

[94] Jaewon Seo, Taewhan Kim, and N. D. Dutt. Optimal integration of inter-task and intra-task dynamic voltage scaling techniques for hard real-time applications. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 450–455, Washington, DC, USA, 2005. IEEE Computer Society.

[95] Hazim Shafi, Patrick J. Bohrer, James Phelan, Cosmin Rusu, and James L. Peterson. Design and validation of a performance and power simulator for powerpc systems. *IBM Journal of Research and Development*, 47(5-6):641–652, 2003.

[96] Shelby Hyatt Funk. *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North-Carolina, Chapel Hill, 2004.

[97] Dongkun Shin and Jihong Kim. Intra-task voltage scheduling on dvs-enabled hard real-time systems. *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1530–1549, November 2006.

[98] Dongkun Shin and Jihong Kim. Optimizing intratask voltage scheduling using profile and data-flow information. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):369–385, 2007.

[99] Dongkun Shin, Hojun Shim, Yongsoo Joo, Han-Saem Yun, Jihong Kim, and Naehyuck Chang. Energy-monitoring tool for low-power embedded programs. *IEEE Design & Test of Computers*, 19(4):7–17, 2002.

[100] Wen-Tsong Shiue. Accurate power estimation for cmos circuits. In *Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology TENCON*, pages 829–833, Texas, USA, 2001.

[101] Anshul Singh and Scott C. Smith. Using a VHDL testbench for transistor-level simulation and energy calculation. In *Proceedings of the 2005 International Conference on Computer Design CDES*, pages 115–121, Las Vegas, Nevada, USA, 2005.

[102] Pushkar Singh and Vinay Chinta. Using probabilistic model checking for dynamic power management. *In Survey report of the University of Illinois, Chicago (ECE Department)*, 2008.

[103] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *Proceedings of the 2005 annual ACM SIGAda international conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies*, SigAda '05, pages 1–10, New York, NY, USA, 2005. ACM.

[104] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware computer systems: Opportunities and challenges. *IEEE Micro*, 23(6):52–61, 2003.

[105] Tat Kee Tan, Anand Raghunathan, and Niraj K. Jha. Embedded operating system energy analysis and macro-modeling. In *International Conference on Computer Design*, pages 515–520, Freiburg, Germany, 2002.

[106] SEI AADL Team. OSATE: An extensible source AADL tool environment. Technical report, December 2004.

[107] Ellidiss Technologies. ADELE: a versatile system architecture graphical editor based on AADL, 2007. http://gforge.enseeiht.fr/projects/adele/.

[108] The SysML Partners. The official OMG Marte web site, May 2007. http://www.omgmarte.org.

[109] Duong Tran, Kyung Ki Kim, and Yong-Bin Kim. Power estimation in digital CMOS VLSI chips. In *Proceedings of the Instrumentation and Measurement Technology Conference, IMTC 2005.*, pages 317– 321, 2005.

[110] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS)*, page 4, San-Diego, California, USA, 2007.

[111] Jim Turley. Operating systems on the rise. *Embedded Systems Design*, 2006.

[112] Roberto Varona-Gomez and Eugenio Villar. Aadl simulation and performance analysis in systemc. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 323–328, Washington, DC, USA, 2009. IEEE Computer Society.

[113] Stephen R. Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, March 1995.

[114] Weixun Wang and Prabhat Mishra. Predvs: preemptive dynamic voltage scaling for real-time systems using approximation scheme. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 705–710, New York, NY, USA, 2010. ACM.

[115] Yue Wang and Nagarajan Ranganathan. An instruction-level energy estimation and optimization methodology for GPU. In *Proceedings of the 2011 IEEE 11th International Conference on Computer and Information Technology*, CIT '11, pages 621–628, Washington, DC, USA, 2011. IEEE Computer Society.

[116] Chien-Chung Yang, Kuochen Wang, Ming-Ham Lin, and Pochun Lin. Energy efficient intra-task dynamic voltage scaling for realistic CPUs of mobile devices. *Journal of Information Science and Engineering*, 25(1):251–272, 2009.

[117] Xia Zhao, Yao Guo, Hua Wang, and Xiangqun Chen. Fine-grained energy estimation and optimization of embedded operating systems. In *Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia (ICESS2008)*, pages 90–95, Chengdu, Sichuan, China, 2008.

List of Publications

International Journals

1. **Bassem Ouni**, Cécile Belleudy, Eric Senn
   Accurate energy characterization of OS services in embedded systems, EURASIP Journal on Embedded Systems, July 2012.

2. Ikbel Belaid, **Bassem Ouni**, Fabrice Muller, Maher Benjemaa
   Complete and Approximate Methods for Off-Line Placement of Hardware Tasks on Reconfigurable Devices, Journal of Circuits, Systems, and Computers, JCSC Vol. 22, No. 2, February 2013.

International Conferences

1. **Bassem Ouni**, Hajer Ben Rekhissa, Cécile Belleudy
   Inter-process communication energy estimation through AADL modeling, SMACD '12, Proceedings of International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design, Seville, Spain, September 2012.

2. **Bassem Ouni**, Cécile Belleudy, Eric Senn
   Energy characterization and classification of embedded operating system services, DSD '12, Proceedings of the 15th Euromicro Conference on Digital System Design, Izmir, Turkey, September 2012.

3. **Bassem Ouni**, Cécile Belleudy, Hajer Ben Rekhissa, Eric Senn
   Energy leakage in low power embedded operating systems using DVFS policy, IEEE FTFC '12, Proceedings of the 11th Edition of IEEE Faible Tension Faible Consommation, Paris, France, June 2012.

4. **Bassem Ouni**, Cécile Belleudy, Eric Senn
   Realistic energy modeling of scheduling, interprocess-communication and context switch routines, DTIS '12, Proceedings of the 7th International conference on Design and Technology of Integrated systems in Nanoscale Era , Gammarth, Tunisia, May 2012.

5. **Bassem Ouni**, Cécile Belleudy, Sébastien Bilavarn, Eric Senn
   Embedded operating systems energy overhead, DASIP '11, Proceedings of the Conference on Design and Architectures for Signal and Image Processing, Tampere, Finland, November 2011.

6. **Bassem Ouni**, Ikbel Belaid, Fabrice Muller, Maher Benjemaa
Placement of hardware tasks on FPGA using the bees algorithm, PECCS '11, Proceedings of the conference on Pervasive and embedded computing and communication systems, Algarve, Portugal, March 2011.

National Conferences

1. **Bassem Ouni**, Cécile Belleudy, Eric Senn
Context switch routines energy characterization, 2 pages, GDR SoC-SiP (System On Chip - System In Package), Paris, France, June 2012.

2. **Bassem Ouni**, Cécile Belleudy, Eric Senn
embedded OS services power and energy consumption, METHODICA '11, 7 th workshop on Methods for the Adaptive Distributed Software, Douz, Tunisia, December 2011.

3. **Bassem Ouni**, Hajer Ben Rekhissa, Cécile Belleudy, Eric Senn
Approach for modeling embedded operating systems energy characterization, 2 pages, GDR SoC-SiP (System On Chip - System In Package) , Lyon, France, June 2011.

4. **Bassem Ouni**, Fabrice Muller, Maher Benjemaa
Placement et ordonnancement des tâches matérielles sur des zones reconfigurables en utilisant le Bees algorithm, 2 pages, GDR SoC-SiP (System On Chip - System In Package) , Paris-Orsay, France, June 2009.

Résumé

La consommation énergétique est devenue un problème majeur dans la conception des systèmes aussi bien d'un point de vue de la fiabilité des circuits que de l'autonomie d'un équipement embarqué. Cette thèse vise à caractériser et modéliser le coût énergétique du système d'exploitation (OS) embarqué en vue d'explorer des solutions faibles consommation. La première contribution consiste à définir une approche globale de modélisation de la consommation des services de base de l'OS: la stimulation de l'exécution de ces services, tels que le changement de contexte, l'ordonnancement et la communication interprocessus, est effectuée à travers des programmes de test adéquats. Sur la base de mesures de la consommation d'énergie sur la carte OMAP35x EVM, des paramètres pertinents soit matériels soit logiciels ont été identifiés pour en déduire des modèles de consommation. Dans une seconde étape, la prise en compte de ces paramètres doit intervenir au plus haut niveau de la conception. L'objectif sera d'exploiter les fonctionnalités offertes par un langage de modélisation et d'analyse architecturale AADL tout en modélisant les aspects logiciel et matériel en vue d'estimer la consommation d'énergie. Ensuite, les modèles énergétiques de l'OS ont été intégrés dans un simulateur multiprocesseur de politique d'ordonnancement STORM afin d'identifier la consommation de l'OS et ceci pour des politiques d'ordonnancement mettant en oeuvre des techniques de réduction de la consommation tel que le DVFS et le DPM. Enfin, la définition et vérification de certaines contraintes temps-réel et énergétiques ont été effectuées avec des langages de spécification de contraintes (QAML, RDAL).

Mots-Clés: Systèmes Embarqués, Consommation d'énergie, Systèmes d'exploitation embarqués, Algorithmes et techniques de réduction de la consommation énergétique, Modélisation AADL, Vérification des contraintes.

Abstract

The ever-increasing complexity of embedded systems that are developing their computation performances poses a great challenge for embedded systems designers: power and energy consumption. This thesis focuses on power and energy characterization, modeling, estimation of embedded operating systems (OS) energy consumption. First, an OS energy consumption characterization flow is introduced: a set of benchmarks, which are test programs that stimulate each OS service separately, are implemented. These programs are executed on the hardware platform: OMAP 35x EVM board. Based on hardware measurements, several hardware and software parameters that influence the OS power/energy consumption are identified and energy consumption mathematical models are extracted. The second contribution consists in proposing a high level model of software application, the OS services and hardware platform using an architecture analysis and design language (AADL). Then, AADL and mathematical models of OS services energy consumption are integrated in a multiprocessor scheduling simulator (STORM) in order to evaluate the OS energy overhead when using DPM and DVFS low power techniques. Finally, a flow of definition and verification of system requirements when allocating application tasks to the processors is proposed. Using a set of languages, RDAL and QAML, various real time and energetic constraints are checked when exploring the design.

Keywords: Embedded systems, Energy consumption, Embedded operating systems, Power management techniques and algorithms, AADL modeling, Constraints verification.