



DEPARTMENT OF ELECTRICAL AND  
COMPUTER ENGINEERING

TECHNISCHE UNIVERSITÄT MÜNCHEN

Ph.D Dissertation

**Pre-Silicon Power and Performance Estimation  
and Optimization for System Scenarios of  
Mobile Communication Platforms**

**Muhammad Mudussir Ayub**





Fakultät für Elektrotechnik und Informationstechnik

Pre-Silicon Power and Performance Estimation and Optimization for System Scenarios of Mobile Communication Platforms

Muhammad Mudussir Ayub

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.) genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing habil. Erwin Biebl

Prüfer\*innen der Dissertation:

1. Prof. Dr. rer. nat. Franz Kreupl
2. apl. Prof. Dr.-Ing habil. Helmut Gräb

Die Dissertation wurde am 16.06.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 14.10.2021 angenommen.

## Acknowledgments

It gives me immense pleasure to write this acknowledgment note as it marks the end of a chapter in my life. A chapter that has great significance not only in my life but in the life of all the people who have helped me write it.

I can not be thankful enough to Allah Almighty, who gave me the strength, courage, and motivation to go through the last five years' ups and downs.

I would begin by thanking Professor Franz Kreupl for giving me this opportunity and supporting my research work. A big thanks to my Industry supervisor Josef Eckmuller, who initially discussed the idea and has played an important role in materializing it.

This work was funded by Intel Deutschland GmbH. I would like to sincerely thank my managers at Intel, Guenther Liebel and Aditya Tomar, for providing me all the resources and a suitable environment to carry out this work.

Many people have played a significant role in helping me achieve this milestone. None of them has played a more vital role than my parents and especially my mother. It is her unconditional love, guidance, and support, that have forged me into a person who has faced a doctorate's hardship with a smile on his face.

# Abstract

5G brings innovation challenges for the spectra of computing and connectivity technology. Cellular modems, in particular, have to support various services and applications requiring low to high data rates. Therefore, the co-optimization of power and performance for architecture along circuit design is inevitable. The design space exploration (DSE) of novel architecture design to enhance the key performance indicators (KPI), i.e., power and performance, for multiprocessors-system-on-chips (MPSoC) is an active research paradigm. The existing methodologies lack a holistic framework to incorporate the performance and power models in the complete design cycle, starting from functional validation to micro-architecture exploration, then to software implementation, and finally to hardware development. Developing a generic methodology for novel and sophisticated embedded systems that incorporate the power and performance model in the complete design cycle for the design space exploration is the ultimate goal of this dissertation. This work presents a modular and distributed setup at electronic system level (ESL) to meet this challenge. It first separates functionality from architecture, and second, the way a power database is attached and integrated makes it practical.

Throughout the years, the complexity, heterogeneity, and scope of modern embedded systems have enormously increased. Rapid exploration of the ample design space for different design choices, such as register transfer level (RTL), has become more challenging and time-consuming at low abstraction levels. The widely-adopted trend in literature and industry is to increase the level of design abstraction into system level to cope with design space exploration challenges. In this dissertation, a methodology is developed, which allows analysis and predicting the power and performance of a heterogeneous MPSoC in the early phases of design at the system level. Moreover, it presents the detailed implementation of this methodology in SystemC and its usage to model the workload of an abstract application and analyze the under-design architecture's power and performance for different mapping scenarios of the application tasks. The principle of separation of concerns is exercised in its true spirit and enhanced for power evaluation by proposing a specific mapping step between power and performance models. Power modeling is carried out in an external power modeling framework, vital for modular setup and rapid prototyping. A use case for the network process is evaluated to demonstrate the effectiveness of the methodology. Results are promising in terms of estimation accuracy and simulation speed achieved.

# Contents

<b>Acknowledgments</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Embedded Systems . . . . .	1
1.1.1 Mobile Communication or Cellular Platforms . . . . .	2
1.2 Problem Statement . . . . .	3
1.3 Positioning of the Proposed Methodology Alongside State of the Art . . . . .	4
<b>2 Fundamentals and State of the Art</b>	<b>6</b>
2.1 Background . . . . .	6
2.1.1 Electronics System Level (ESL) . . . . .	6
2.1.2 Modeling of Embedded Systems . . . . .	9
2.1.3 Transaction Level Modeling . . . . .	10
2.1.4 Power Consumption of Embedded Systems . . . . .	11
2.1.5 Power Management Techniques . . . . .	12
2.1.6 Power Analysis Approaches and Methods . . . . .	13
2.1.7 Power Model Development . . . . .	14
2.2 Related Work . . . . .	16
2.2.1 ESL Performance Estimation Frameworks . . . . .	16
2.2.2 ESL Power Estimation Frameworks . . . . .	21
2.3 Fundamentals . . . . .	24
2.3.1 Architectural Abstraction Level . . . . .	25
2.3.2 Software Abstraction Level . . . . .	26
2.3.3 Hardware Abstraction Level . . . . .	27
2.3.4 Holistic and Modular Methodology . . . . .	27
<b>3 POEM: Power and Performance Optimization and Exploration Methodology</b>	<b>29</b>
3.1 Application Modeling . . . . .	29
3.1.1 Task Execution Model (Workload) . . . . .	30
3.1.2 Implementation of the Task-Graph in SystemC . . . . .	34
3.2 Architecture Modeling . . . . .	45
3.2.1 Task Scheduling and Resource Allocation . . . . .	45
3.2.2 Virtual-Engine . . . . .	48
3.3 Power Modeling . . . . .	48
3.3.1 Docea™ . . . . .	50

<b>4</b>	<b>Implementation of the Methodology at Architectural Abstraction Level</b>	<b>55</b>
4.1	Methodology: Overview . . . . .	55
4.1.1	Implementation of Application Model . . . . .	57
4.1.2	Implementation of Performance Model . . . . .	58
4.1.3	Implementation of Power Model . . . . .	60
4.1.4	Interface and Mapping . . . . .	61
<b>5</b>	<b>Network Processor Case Study</b>	<b>70</b>
5.1	System level uses of Mobile Communication Platforms . . . . .	70
5.2	Network Processor Use Case . . . . .	71
5.3	Stand-Alone Application Simulation Using Virtual Engines . . . . .	72
5.3.1	Mapping All Tasks on a Single Virtual Engine . . . . .	73
5.3.2	Mapping Tasks on Multiple Virtual Engines . . . . .	75
5.4	Co-simulation of the Application and the Architecture Model . . . . .	79
5.4.1	Mapping Tasks on an Architecture Model With Two CPU Cores . . . . .	79
5.4.2	Mapping Tasks on an Architecture Model With Three CPU Cores . . . . .	82
5.4.3	Increasing the Memory Size and Result Analysis . . . . .	82
5.5	Power and Performance Co-Simulation . . . . .	84
5.5.1	Simulation Environment Settings and Goals . . . . .	86
5.6	Comparison with Platform Architect . . . . .	87
<b>6</b>	<b>Integration of the Methodology At Software and Hardware Abstraction Levels</b>	<b>93</b>
6.1	POEM at Software Abstraction Level . . . . .	93
6.2	POEM at Hardware Abstraction Level . . . . .	94
<b>7</b>	<b>Conclusion and Future Work Directions</b>	<b>98</b>
7.1	Conclusion . . . . .	98
7.2	Future Work Directions . . . . .	98
	<b>List of Figures</b>	<b>100</b>
	<b>List of Tables</b>	<b>104</b>
	<b>Listings</b>	<b>105</b>
	<b>Bibliography</b>	<b>106</b>
	<b>Acronyms</b>	<b>113</b>
	<b>Publications of the Author</b>	<b>116</b>

# 1 Introduction

This chapter intends to give the reader an overview and motivation behind this work. How a shift towards sophisticated embedded systems design and specifically mobile platforms complexity in the light of 5G and beyond drives to co-optimize power and performance. The challenges and the current practices of power and performance analysis and optimization during the project life span of an embedded system are critically looked at to determine the potential gaps for co-optimizations. In the end, the positioning of the proposed methodology, alongside state of the art, is also introduced to showcase the work's contribution.

## 1.1 Embedded Systems

Embedded systems are becoming an integral part of our life as they have brought enormous convenience in many aspects, such as communication, security, health issues, and automation. The modern-day electronic devices (i.e., mobile phones, tablets, smartwatches) are based on heterogeneous multiprocessors system on chip (MPSoC) architecture. These architectures range from hierarchical memory systems to fully programmable processors to dedicated hardware accelerators for time-critical application tasks.

As services and applications that users expect to run on these electronic devices are becoming more sophisticated and resource-hungry [1] (e.g., high-definition games and video encoding/decoding, with several new demanding services approaching with 5G networks on the horizon), so multiprocessors system on chips are becoming more and more complex. To overcome the complexity and to manage the limited available resources efficiently, there are two well-known approaches.

The first option is to increase the frequency of the processor to achieve a higher number of computations in a given clock cycle. The second promising solution is to exercise the parallelism, divide a program or task into independent executable parts, and schedule them on different resources in multiprocessors system on chips. These two approaches help to improve the max throughput or performance of the system. However, an increase in frequency raises thermal concerns and causes higher power dissipation. Similarly, to exploit parallelism in multiprocessors system on chips has a cost in terms of the power budget.

Hence, most of the time, performance increase comes at the cost of the power budget. Performance and power are conflicting parameters in the silicon design regime. Therefore, power modeling and optimization techniques are as critical in the system design of multiprocessors system on chips as performance, given the fact that most of the multiprocessors system on chips have a limited power budget in the form of battery capacity. Moreover, a user expects to run these devices for a significant amount of time (i.e., a day or two) without worrying

about connecting them to the power source.

The decrease in device size and the quest to put more and more transistors in a given area (nano-miniaturization) following Moore's law [2] is another reason to optimize the increase in static power consumption. As the recent study for mobile platforms shows that two factors restrict the maximum receiving rate in smartphones. One factor is the computation capability of the base-band processor, and the other is heat dissipation [3].

### 1.1.1 Mobile Communication or Cellular Platforms

To give a perspective about how relevant cellular platforms are in today's day of age, we need to look at the statistics provided by GSMA Intelligence [4]. "According to GSMA real-time intelligence data, there are now over 5.15 Billion people with mobile devices worldwide – This means that 66.60% of the world's population has a mobile device (cell phone, tablet, or cellular-enabled IoT devices)". Moreover, these numbers are increasing with every passing day. Therefore, the optimal design of the Mobile communication platform is one of the major differentiating factors for the users to prefer one supplier from another.

Cellular platforms generally have two processors; application processor (AP) and a base-band processor, aka cellular modem. At a high abstraction level, the application processor is responsible for running all the diverse and demanding applications alongside the operating system on the cellular platform. Similarly, the task of the cellular modem is to establish a stable connection with the network and send and receive data at a specific rate.

The 3rd generation partnership project (3GPP), responsible for standardization of mobile communications (i.e., 2G up to 5G ), is a worldwide cooperation of standardization bodies. The 3GPP standards are evolving with every passing year, and the cellular modem has to adapt accordingly. Likewise, multimedia applications are becoming more and more resource hungry; as a result, more sophisticated design and scheduling of applications are expected from application processor. Therefore, 3GPP standards and multimedia are driving power and performance issues in cellular platforms. However, a recent study shows that the co-optimization of power and performance for novel and sophisticated embedded systems such as mobile platforms is a way forward [5].

### 5G and Challenges

5G will revolutionize the world; connected devices, self-driven cars, and cellular modems supporting much higher data rates promise an exciting future ahead. The materialization of the connected devices, i.e., machine-to-machine or device-to-device communication, has vast potential but still needs time to mature. The task at hand is to develop efficient 5G cellular platforms. They belong to the category of multiprocessors system on chips, which has massive potential for design space exploration as the world is riding the tide of higher data rates for 5G and beyond.

The high data rate performance criteria in cellular platforms will come at the cost of a considerable increase in power consumption. This challenge has opened the door to explore new configurations and architecture designs to meet the performance requirements with



minimum overall power consumption. Hence, the design space exploration of innovative architectural design to enhance power and performance for mobile communication platforms is a current research paradigm.

Design space exploration and power analysis of the cellular modem are vital because not only do they have to meet evolving 3GPP standards, but they also have to support legacy use cases. For example, a new architectural design of the cellular modem may suit well for 5G data rates. However, if the power budget for legacy use cases (2G, 3G) is not the same or less, we have a design challenge to resolve here.

In mobile communication platforms, a few key performance indicators (KPIs) allow an easy comparison of different platforms. KPIs are metrics like data rate, area, and power. Power is one of the most relevant KPIs in mobile communication by nature. Therefore, we have to consider the trade-off between power and performance/data rates in this work. Furthermore, the area of the mobile communication platform is highly dependent upon the technology node being used, system specifications, both of these are indeed for our work. Thus power and performance trade-off is carried out without considering areas as an objective in this work.

## 1.2 Problem Statement

Post-silicon KPIs estimation and optimization for mobile communication platforms or other complicated multiprocessors system on chips are not attractive anymore, as the time to market is squeezing, the efforts and time required to incorporate changes (design iterations), and optimization margins are not significant enough at this point. Pre-silicon functional and non-functional (i.e., power and performance) verification and optimization can enable a shift left in the timeline of the embedded system design cycle. A correct decision taken at the very beginning of the design cycle will have more impact than a decision taken later time in point.

Along with pre and post-silicon, selecting the right abstraction level is critical for key performance indicator estimation and optimizations. A mobile user runs a single or set of applications from a plethora of applications on it. For such heterogeneous multiprocessors system on chips, the system level power and performance estimation is required for specific system use cases or scenarios. For example, in a mobile platform, a relevant question might be the power consumption in the activation of triple carrier aggregation in 4G.

Power estimation and optimization need to be addressed at the architecture level, software (SW) implementation level, and hardware (HW) implementation level. However, in industry, the power analysis and estimation at the different levels are not well coupled. As a result, power models development effort, accuracy, and simulation speed, along with their integration to the system level, is a complicated task.

Power estimation at the architecture level is required to influence architecture and design decisions. However, it is challenging because it has to be done in a very early stage of the design process, based on uncertain data and with limited resources and effort.

At the software level, we have to analyze and minimize the power consumption based on software running on a fixed hardware architecture. The pre-silicon phase requires an

abstracted hardware architecture model to run the software with good simulation performance and timing approximation needed for reliable power estimations. The goal of this step is to do the power optimization of the software.

At the hardware implementation level, we usually analyze the power consumption of hardware libraries, building blocks, and hardware modules. The result of that low-level power estimations will also be used as input for power estimation at software and architecture levels.

The other non-functional property performance, like power, also needs to be addressed at the abstraction mentioned above levels in the coupled way; the challenging part is that the system level use cases for power and performance are different. For example, for performance, the system use cases targeting the max-throughput and latency of the components are under the radar. Such system use cases are not very helpful for power optimizations.

In summary, there is a need to work towards a holistic methodology for functional and non-functional properties estimation and for the design space exploration to analyze different variants and configurations of the heterogeneous architecture of mobile communication platforms. Moreover, further architecture refinement through regression and continuous feedback is required. Potential application mapping to evolving architecture according to the demanding standards can be achieved in the best possible manner regarding power and performance KPIs.

### **1.3 Positioning of the Proposed Methodology Alongside State of the Art**

To perform an effective power performance trade-off analysis in early design phases, the following key ingredients are required:

- Abstract performance model of the hardware architecture candidate(s)
- High-level description of the use cases under analysis
- Power model for the relevant hardware components

In real projects, these different parts are developed by different teams. Hardware architects define different variants of the overall architecture. System architects determine the high-level structure of the system use cases (functional elements and their interaction). Power experts do the initial power analysis, defining early abstractions of power models at the component level, derived from previous generations or high-level estimates, and including all relevant technical parameters.

The main contribution of this work is a novel power optimization and exploration methodology (POEM), which addresses such distributed project setup by using the concept of separation of concerns [6].

The first step allows independent power and performance analysis of the application, and in the second step, it combines power and performance view to carry out the optimization. Thus,

this modular setup helps to understand better, extend, and maintain a complex embedded system.

POEM describes applications at different abstraction and accuracy levels. Moreover, this methodology is applied for novel 5G use cases.

The differentiation of this work is mainly implemented in the following components:

1. **Performance modeling** A productivity library on top of Accelera SystemC is used to enable the loosely-timed and approximately-timed simulation models. This setup helps POEM to cover application scenarios for power and performance analysis holistically.
2. **Application modeling:** The activity or the functionality modeling at different hierarchy levels (firmware, OS, application) with the help of task graphs or running the actual software on the simulation platform.
3. **Power modeling:** Generation of comprehensive power database, hierarchical in nature and generic enough to support heterogeneous functional stimulus.
4. **Connecting power and performance modeling:** POEM simulates application and performance models in a SystemC kernel with the help of a configuration file and maps the generated output stimuli (i.e., functional states, frequency, and usage) to a power model. Thus, POEM supports dynamic power management techniques like dynamic voltage and frequency scaling (DVFS), power gating, and clock gating.

## 2 Fundamentals and State of the Art

Chapter 1 provides an overview of the proposed methodology and why it is needed. This chapter describes the methodology's fundamentals and introduces important terminologies throughout this dissertation to understand, identify, and shed light on different embedded system project phases. State of the art is also discussed to differentiate and highlight the importance of the work done.

### 2.1 Background

#### 2.1.1 Electronics System Level (ESL)

Software and hardware development run in parallel in the conventional design flow of the embedded system. The design specifications from a customer drive this process. Functional and extra-functional requirements are the two broad categories of these specifications. Functional requirements specify; *what the system does?* e.g., a cellular modem should be capable of making calls, sending text messages, and running different applications.

Similarly, extra-functional requirements define; *how (good or bad) the system does?* e.g., a power consumption rate of the cellular modem, time taken to open an application, time to market, cost of the phone, and the final product area. Design space exploration for power and performance requires modeling of the embedded system for extra-functional requirements along with functional requirements.

The work done by Gajski and Kuhn is known as Y-chart [7], describes different abstraction levels at which a system can be modeled. Y-chart consists of five concentric circles in three different domains - behavioral, structural, and physical, as shown in Figure 2.1.

System modeling started with describing an embedded system manually a few decades ago; as the embedded system's complexity and functionality gained momentum, the manual modeling was no longer possible. It moved up from the physical level to gate-level and from the gate-level to the register transfer level (RTL). Well known, VHDL and Verilog are examples of register transfer level descriptive languages.

The design space of embedded systems, especially cellular modems, is vast. The register transfer level abstraction level is not the appropriate level to model such a complex system as the modeling effort and simulation time required are very high.

Electronics system level is the highest level of abstraction possible for modeling of complex multiprocessors system on chips extra-functional properties. The decisions made at this level will impact the entire design space [8]. The challenge faced at electronics system level is little to no information about the low-level implementation details. Figure 2.2 shows the

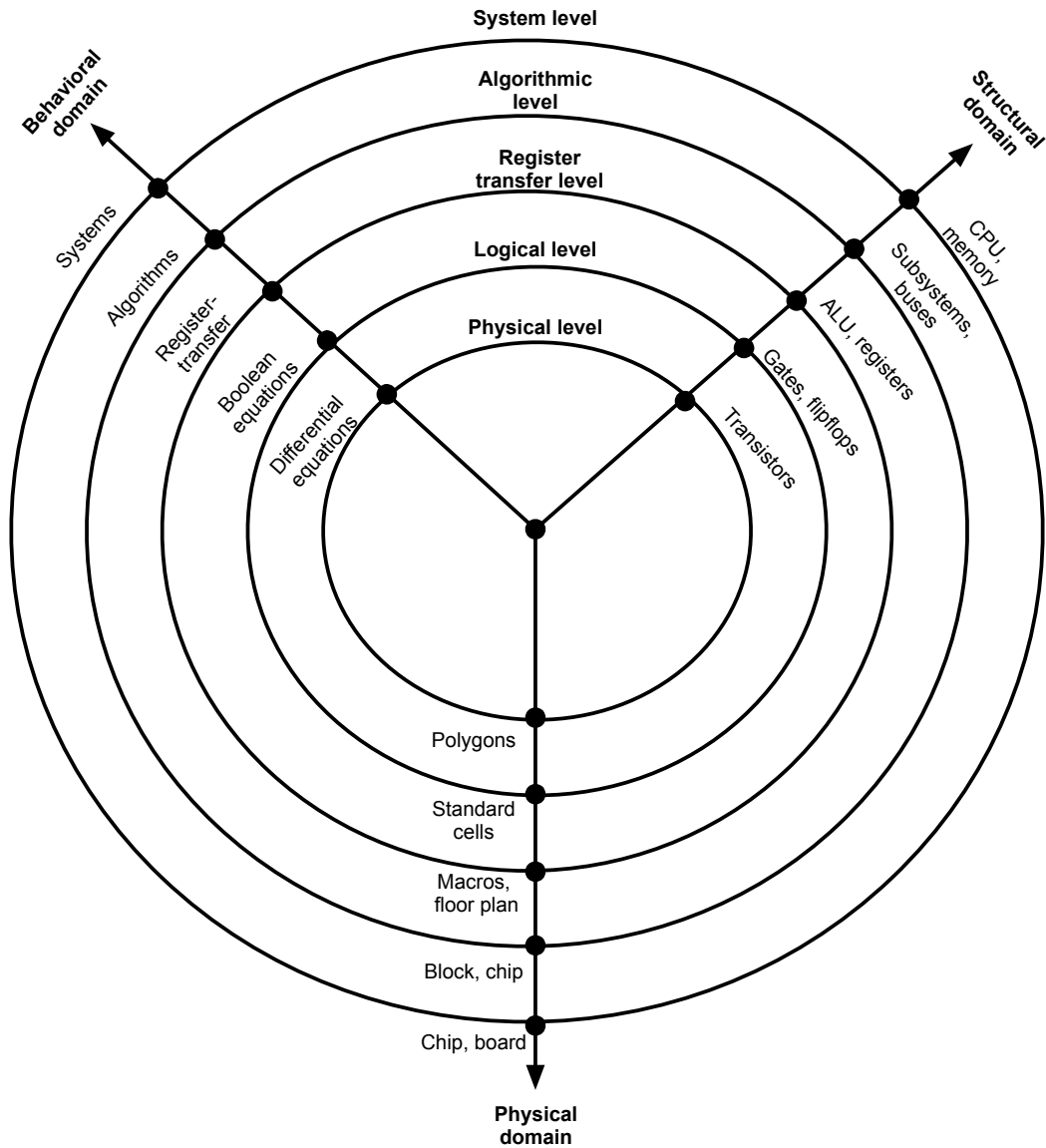


Figure 2.1: Y-chart shown as a concentric circles, representing different abstraction level in three different domains [7]. The innermost circle corresponds to the lowest abstraction level, and the abstraction level increases as the distance of the concentric circles from the center increases. Fast and accurate system modeling of heterogeneous, multicore system on chip needs to be done at the highest abstraction level: system level. It corresponds to actual chip or prototype boards in the physical domain, systems in the behavioral domain (even above algorithms), and CPU, memory blocks in the structural domain.

design flow for embedded systems and how the simulation time, design details, and design abstraction varies along the complete development cycle.

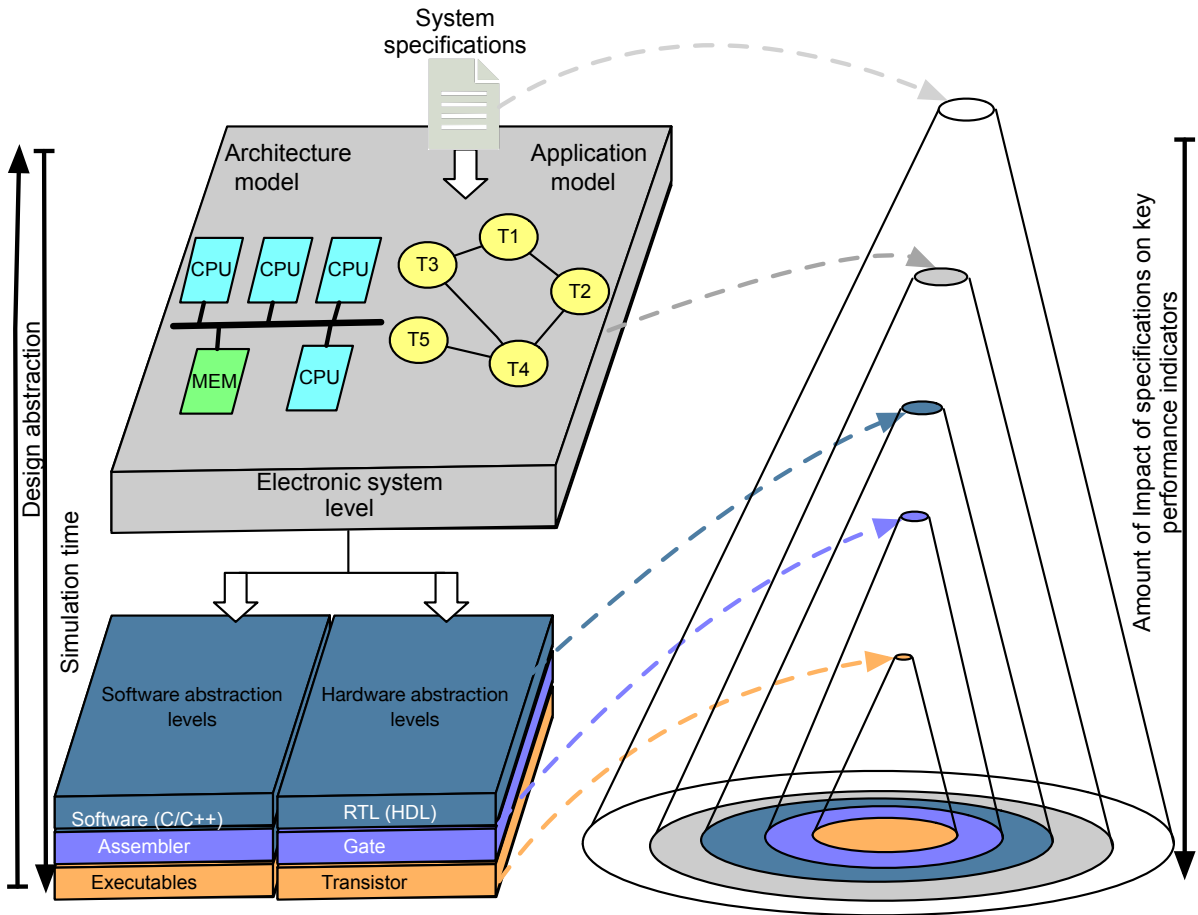


Figure 2.2: Contemporary design flow of an embedded system and an overview of the relation between design abstraction, design detail and simulation time: The embedded system is divided into four abstraction levels. Simultaneously, design specifications sit at the top, and the system level, also known as the electronics system level (ESL), is the first abstraction level derived based on customer-provided design specifications. The abstraction level is the highest, the time required for a system level use case is the lowest, and very few design details are known at this point. The hierarchy of the funnels is to visualize the available field of design space exploration. The complete design space exploration field is accessible from the top where the design specification resides. These streams are further divided into three abstraction levels. Register transfer level Design is the first, and the transistor level is the last abstraction level in the hardware stream. Similarly, software written in C/C++ is the first, and executable is the last abstraction level in the software stream.

Electronics system level offers more than one way of modeling an embedded system; therefore, this abstraction level, unlike the register transfer level and other lower abstraction levels, does not have a clear, commonly agreed definition. According to [9], it can be described as "the utilization of appropriate abstractions in order to increase comprehension about a system, increase the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints."

The cycle-accurate (CA) modeling and transaction-level modeling (TLM) approaches were explored above RTL to raise the abstraction further to the electronics system level. Cycle-accurate modeling did not emerge as a winner because of multiple reasons, as mentioned in [10]. Higher modeling efforts (almost the same as register transfer level), 10x lower simulation speed than anticipated, register transfer level annotation for cycle-accurate characterization due to tight deadlines towards the end of the embedded system development made it impractical. Liberating the system design from synthesis-related constraints was one of cycle-accurate's benefits, resulting in a rise in abstraction.

### 2.1.2 Modeling of Embedded Systems

An embedded system at the electronics system level consists of architectural resources like processors, memories, hardware accelerators, and interconnects in the forms of buses and Network-on-Chips (NoC). TLM abstracts from the low-level implementation details required for register transfer level and models the architecture resources' functionality and their communication separately. As the name suggests, the communication between system resources happens in the form of transactions. Thus the simulation speed is much faster than register transfer level and cycle-accurate.

TLM is carried out with a high-level language capable of software development and hardware modeling at a conceptual level without requiring implementation details. Such high-level languages include SystemC [11], SpecC [12], System Verilog [13] Hpasal [14], and Hardware-C [15].

Since the last decade, SystemC has become a de facto language to model an embedded system at the electronics system level. It is partially because of the backing of big electronic design automation (EDA) companies such as Synopsys [16] and Cadence [17]. Due to multiple reasons, SystemC stands out from the rest of the above mentioned high-level languages. We will first describe the language briefly, and a few advantages will follow.

SystemC is a language that consists of C++ class libraries and macros. SystemC has an event-driven simulation kernel. It has all the data types of C++, and besides C++ data types, few additional data types, i.e., *sc\_time* and user-defined data types, are also part of it. SystemC data types use *sc* as a prefix, so an initialization of an integer variable in C++ takes the form of *sc\_int* in it.

SystemC models the system in modules; a module can represent a process written in plain C++ and running concurrently with other modules/processes. Signals of all data types can be used for communication purposes during the event-driven simulation. Modules may consist of port, method, and channels for connectivity. There are two types of process in SystemC, *SC\_METHOD*, and *SC\_THREAD*. *SC\_METHOD* is suitable for pure functional description

(without timing), and *SC\_TREAD* process has timing notion because of the introduction of *wait()* function. System modeling in SystemC comprises of two phases, elaboration and simulation. In the elaboration phase, design objects are instantiated, such as methods, ports, and channels, and during the simulation phase, the system model is being run.

Along with the embedded system modeling at electronics system level, SystemC can also be used for functional verification, architecture exploration, performance modeling, software development, and high-level synthesis of the embedded system using different transactional's abstraction modeling.

To understand the different transactional abstraction models, i.e., loosely-timed (LT) and approximately-timed (AT) in SystemC, as mentioned in [18, 10, 19], consider the example of software development and performance modeling. Software development depends upon a register, and a data-accurate view of the system, simulation speed, and early evaluation is more critical for software refinement than timing information of the simulation. For performance analysis, accurate timing information is essential; thus, modeling micro-architectural details of key modules is required.

Loosely-timed and approximately-timed were previously known as programmers view (PV) and programmers view with timing (PVT), respectively. Loosely-timed has enough timing information to boot the operating system (OS) and run the multi-core system. In loosely-timed, processes can run ahead of simulation time, and a transaction has only two points. The first point is the start of the transaction from a Master module, and the second point is the end of the transaction to a Slave module. Loosely-timed modeling can not have pipe-lined transactions; hence, suitable for software development that does not depend on timing accuracy.

Architectural exploration by adding the system's micro-architectural details is possible with the help of approximately-timed abstraction modeling. In approximately-timed abstraction, processes run in a lock setup. Furthermore, each transaction has four points instead of two-point. Begin and end of transaction request and begin and end of the transaction response. Approximately-timed also allows blocking, non-blocking, and pipe-lined transactions, making it suitable for performance analysis of the embedded system.

### 2.1.3 Transaction Level Modeling

The open SystemC initiative (OSCI) TLM 2.0 [20] present a standardized approach to use TLM. TLM 2.0 is a tool kit based on a set of library functions written in SystemC and suitable for functional validation, especially for multiprocessors system on chip (MPSoC) simulation. SystemC standards have integrated the TLM 2.0 transactional modeling since 2011.

TLM 2.0 separates the computation details from communication details as the communication and computation's internal implementation details are hidden. Each can be designed at different abstraction levels. They communicate with each other through interfaces using transactions. A transaction is an aggregation or encapsulation of several events, signal states, and data exchange details into a single function call.

Advantages of TLM over register transfer level and cycle-accurate abstraction levels are the following:



- One of the most significant benefits of TLM is providing a simulation environment in which hardware and software can be modeled as concurrent running processes. That results in efficient hardware-software co-design [10, 19].
- TLM resolves synchronization issues that designers face during hardware-software co-design as a homogeneous environment is available for hardware-software co-design [21]. Beneath the TLM abstraction level, hardware and software are modeled in different simulation environments. For their co-design, Inter-Process Communication overhead is replaced with threads switch over in TLM and simplifies the synchronization.
- TLM abstracts away low-level implementation details, and it provides modeling speed-up compared to the cycle-accurate and the register transfer level. A 10x modeling speed-up was reported compared to the register transfer level abstraction level, and a 7x modeling speed-up was reported compared to the cycle-accurate abstraction level in [10, 19].
- The simulation speed of TLM is also significantly higher than in register transfer level and cycle-accurate abstraction level [10].

#### 2.1.4 Power Consumption of Embedded Systems

The embedded system targeted in this work is multiprocessor system on chip (MPSoCs), i.e., a cellular modem. Such multiprocessors system on chips are built using CMOS technology. An inverter is the most simple circuit of CMOS, and it is made of a *pull-up network* (P-type transistor) and a *pull-down network* (N-type transistor). According to [22], CMOS circuits' power consumption can be divided into two types, as shown in equation 2.1.

$$P_{total} = P_{static} + P_{dynamic}, \quad (2.1)$$

where  $P_{total}$  is the total consumed power,  $P_{static}$  is the static power and  $P_{dynamic}$  is the dynamic power.

Four major contributors to these two types are *leakage power*, *other static power*, *short-circuit power*, and *switching power*. The two contributors to static power consumption that are independent of the circuit's switching activity are *leakage power* and *other static power*. The parasitic currents cause *leakage power*, and *other static power* is contributed by the additional circuit added and the current it causes to flow between the supply rails. Therefore,  $P_{static}$  consumption is independent of computing and occurs when the embedded system is connected to a power supply.

$$P_{static} = V_{dd} \cdot I_{leakage}, \quad (2.2)$$

where  $V_{dd}$  is the supply voltage, and  $I_{leakage}$  is the current flowing between the supply rails.  $I_{leakage}$  increases with the reduction in transistor size, and  $P_{static}$  that is directly proportional to it also increases.

The *short-circuit power* and *switching power* corresponds to dynamic power consumption  $P_{dynamic}$ .  $P_{dynamic}$  results from the switching activity and in result charging and discharging of the capacitors in a component. The *short-circuit power* consumption caused by the instantaneous short-circuit between the supply voltage ( $V_{dd}$ ) and ground (during the switching of gate state) is negligible compared to *switching power* consumption and, therefore, often left out [23].

$$P_{dynamic} = \alpha \cdot C \cdot V_{dd}^2 \cdot f, \quad (2.3)$$

where  $\alpha$  is activity ratio,  $C$  is the capacitance, and  $f$  is the clock frequency of the component.

### 2.1.5 Power Management Techniques

Power management of a sophisticated embedded system like multiprocessors system on chips is studied since many years because of its obvious need and importance. Two significant contributors to power consumption are voltage and clock frequency. Static power depends on the connected voltage (power supply  $V_{dd}$ ) and the leakage current produced. Dynamic power depends upon voltage, frequency, and an activity factor. Most of the time, Static power was neglected in the past, as its contribution towards total power consumption was negligible. As the embedded system's technology node is shrinking, the contribution of static power towards total power consumption is increasing and can not be neglected anymore.

Power consumption in the embedded system can be managed with software and hardware. Power management policies in the software are implemented in the *power manager*, and the *power controller* (that consists of hardware components) control voltage and clock in hardware to manage the power. The three well-established techniques, according to [24], are power gating, clock gating, and dynamic voltage and frequency scaling. There are other techniques, which are either combination or a slight variant of these three fundamental techniques.

#### Clock Gating

Clock gating [25, 26] is a technique in which a component's clock is turned off if it is not computing; thus, it stops the component's switching activity and reduces dynamic power consumption. A significant portion of the power consumed in the clock tree can be reduced, and many low-level synthesizers can execute it automatically. This technique does not impact static power consumption.

#### Power Gating

Power gating is a technique in which a component not used is power gated, which means switched off. It can reduce static power plus dynamic power. Nevertheless, to switch off a component, not in use for a time interval is not straight forward. First, the power gated component's internal states need to be stored in retention registers before power gating. Second, the power penalty and the time required to restore the component must be carefully managed. In case of a deadlock (if the application requires a component that is yet power

gated), the power penalty is higher than this technique's saved power. Therefore the decision to switch off (power gate) a component or not is non-trivial; it depends on many factors, that include, i.e., the inactive interval, application, firmware, and technology node.

### **Dynamic Voltage and Frequency Scaling (DVFS)**

Dynamic voltage and frequency scaling is the most sophisticated one of all techniques discussed. It impacts both voltage and frequency that are quadratically and directly proportional to power consumption, respectively. Dynamic voltage and frequency scaling is a technique in which a component's clock frequency and voltage are changed dynamically depending upon the data load and processing demand. If a component is performing computation faster than required, the computation rate can be decreased by reducing the clock frequency that results in the decrease of dynamic power consumption. The voltage and frequency pair selection is a task as non-trivial as power gating. Power controller that controls the voltage and clock generator hardware modules and power manager that implements power policies in software works together to choose the right voltage and frequency pair.

### **2.1.6 Power Analysis Approaches and Methods**

Power estimation methodologies can be divided into two main categories, low-level, and high-level methodologies. The accuracy of the power estimation and the time required for executing the methodology are inversely proportional. Low-level methodologies are more accurate but the simulation time is very high. Among the low-level tools worth mentioning are, SPICE [27] Diesel [28] and Petrol [29], which work at transistor, gate and register transfer level (RTL) respectively. It is almost impossible to use such low-level tools for a full system design of complex mobile devices of these days, as the fine circuit details are not available in early design phase and the simulation time required makes them impractical for regressive design space exploration of complex multiprocessors system on chip.

High-level power modeling methodologies can be based on instruction level power analysis (ILPA), a methodology first introduced by V. Tiwari [30] or functional level power analysis (FLPA), proposed by J. Laurent [31]. ILPA divides the program running on a processor into a set of discrete instructions or instruction sequences and assign to them power/current values. The complexity associated with assigning a current consumption to each instruction is by far the biggest drawback of this work [32], alongside missing instantaneous power estimation and the consequences of address and data changes when a program runs. To overcome the deficiencies of ILPA, FLPA and the associated SoftExplorer [33] power modeling tool revolve around the extraction of two types of parameters for any target embedded system, i.e., the architectural and the algorithmic parameters. processing unit (PU), instruction management unit (IMU), buses and memories are examples of the former, whereas operating voltage, frequency, and input data word length of the latter, as they also affect the power consumption of the platform. Although the abstraction level that FLPA propose is rather high, this approach suffers in terms of identification and extraction of the appropriate parameters to simulate the power model in complex platforms.

Power modeling techniques used at electronics system level are analytical or simulation-based. Analytical techniques are static and, most of the time, used for the worst-case and best-case bounds. Such techniques are well suited if the application or system level use case under evaluation has deterministic behavior [34, 35, 36, 37, 38]. These techniques do not require a functional simulation input to derive them, and that is one significant advantage over the simulation-based techniques. One typical example of an Analytical technique is spreadsheet analysis with the help of the Excel tool. Manual input is prone to errors, and in industry, the inability to maintain and share different versions of spreadsheet-based power estimation is not preferred.

Simulation techniques are attractive as they are dynamic and well suited for non-deterministic applications; therefore, power and performance analysis over time help hardware software optimization, i.e., by spreading out the power peaks or scheduling the activities differently. Simulation-based techniques require embedded system modeling and simulation capability. SystemC/TLM modeling discussed in 2.1.3 is the right candidate for such techniques, and many tools and frameworks in the industry use this combination. Our dissertation also uses this technique for power estimation.

### **2.1.7 Power Model Development**

Power state machine and Linear Regression are the most commonly used approaches to develop electronics system level power models. All the electronics system level power estimation methodologies use these standard approaches in one or another way to develop a power model.

#### **Power State Machine**

PSM, very similar to finite state machine (FSM), is used to model the power consumption of an embedded system or subsystems at electronics system level. The characteristics associated with PSM include modeling power consumption and an agnostic approach towards functional simulation. An agnostic approach towards simulation means that the system use case and time notion of the simulated system does not impact how the PSM model the system's consumption. This agnostic approach is in-line with our goal of developing a modular methodology.

The input stimuli of power state machine come from the timed, functional simulation of the embedded system under evaluation, and these input stimuli trigger the different states in PSM. The transition of power states in PSM changes the estimated power consumption. Each power state has a different power budget behind it, and the total power is the sum of all the power states in one instance. Hence, PSM techniques are linked with power estimation only approaches.

In the literature, all the approaches that use PSM to estimate power consumption are similar, with minor differences. In [39], PSM is developed for the system's components, and the power budget of the PSM power states is derived manually. The power budget calculation method

keeps the system's component in a specific state and measures the power for a long enough duration. In [39], annotated power states are later verified with hardware measurements.

The power modeling approach for design space exploration at electronics system level presented in [40] uses PSM. A unique feature is that each component could have a different power state depending upon the global governor, which decides the mode of operation, fast, medium, and slow. This work does not focus on creating the PSM but comparing different configurations of the embedded system for early design space exploration.

PSM concept is also discussed with unified power format (UPF) [41]. UPF is the only standard of IEEE, which provides guidelines to model the power of an embedded system at electronics system level. Power Modeling, with just two power states in PSM, is shown in [42]. The two power states are *Active* and *no operation*; this work also explains how to annotate the power states for critical components like hardware accelerators, memories, and processors.

### Linear Regression

Linear Regression is another frequently used approach to develop power models at electronics system level. This approach can create the power model given the input variables, and power consumption is known benchmark or reference scenarios. Therefore, it is frequently used for black-box modeling; for example, if a processor's internal power states are unknown (as it could come from a vendor or third party), linear regression can develop its power model.

The summation is the basic principle behind linear regression; if an embedded system consists of more than one crucial component, its total power consumption is the sum of power consumed in each component. The power consumption of each component depends upon the activity being performed inside the component.

The linear regression approach models the power as the weighted sum of some input variables. These weighted factors are first extracted from reference scenarios or for a known benchmark. Linear regression is a particular case of PSM; if a linear regression model has variables with binary values, and only one variable must be in the high state at a given time, then the weighted factors correspond to the power annotation of power states in PSM.

The linear regression approach suffers the apparent problem of over-fitting and instability for non-linear components in an embedded system, i.e., a power amplifier. There are multiple solutions proposed to tackle these shortcomings, i.e., uses non-negative values of weighted factors. non-negative least square (NNLS), a constrained linear regression, is proposed in [43] for CPU and GPUs. [44] tackles the over-fitting issue by proposing the tree structure. Each branch of the tree is selected based on control input, and regression equation at a specific branch level of the tree is executed based on data input. Control and Data are the input variable for linear regression.

Several approaches that develop power models using linear regression are discussed in [45, 46]. In [45] number of instructions and registers, type of the instructions, and data inside the registers are the variables, and a manually created reference scenario extracts weighted factors of these variables. In [46], functional units are used as a variable, and weighted factors tell the power consumed by each functional unit depending upon their activity.

A case study for network on chips (NoC) [47] uses linear power models. Input variables

are the switching activity inside the transmitted data and output port and the number of active output ports. The reference power trace to find out the weighted factor is gathered by gate-level simulation.

## 2.2 Related Work

This section will review the electronic system level simulation tools or frameworks available in industry or academia for the embedded system's key performance indicators analysis. As we have seen, TLM can model the performance along with other aspects (functional verification, software development, and high-level synthesis) of the complex multiprocessors system on chips. For early path findings, micro-architecture and performance analysis are the most crucial aspects from the key performance indicators perspective.

The second topic of interest is power-aware architecture modeling at the electronics system level, adding non-functional properties like power information into the embedded system analysis using TLM. In this section, we will review the frameworks addressing power and performance together, which is the focus of this dissertation.

### 2.2.1 ESL Performance Estimation Frameworks

There are various system level design frameworks developed in industry or academia over the last decades. In this section, a selection of these frameworks are summarized. We only review state of the art simulation-based system level performance evaluation methods, which do not require mature application and architecture models (as is the case at electronics system level).

#### **SPADE**

System-level Performance Analysis and Design Space Exploration (SPADE) is a tool and framework for system level architecture exploration of heterogeneous signal processing systems[48, 49]. SPADE mainly focuses on signal processing domain embedded systems, and it uses a trace-driven simulation technique for performance evaluation. The application is modeled using the Kahn process network, and the architecture is constructed using building blocks from a library. The architecture model has generic processing, communication, and memory elements. The performance specifications, such as execution time and latency, are configured using some parameters during architecture instantiation. For this purpose, look-up tables of pre-computed performance parameters from data-sheets or low-level simulations are used to configure each architecture component. During a simulation, each process in the application model, while reading their input FIFOs, processing their local data, or writing to their output FIFOs, generates a trace of symbolic instructions such as read, write or execute. These symbolic instructions represent the workload of the application on the architecture. The trace of each process in the application model is mapped on one of the architecture resources that interprets the corresponding trace's symbolic instructions and simulates their timing and performance behavior according to their performance configurations. However, the symbolic instructions do not contain any inter-process control expressions, and the

architecture resources execute them in the same order as they are received. Therefore, the potential concurrency of the application processes cannot be exploited [50], such as time-dependent behavior. The simulation speed is very fast because the architecture resources simulate the symbolic instructions without processing the actual data.

### **ARCHER**

It is a framework for system level performance evaluation and architecture exploration of heterogeneous embedded systems, and it focuses on streaming-based applications [50, 51]. Application is modeled as KPN with non-determinism modeling capabilities of the YAPI tool [52]. ARCHER extends the SPADE framework by using symbolic programs instead of symbolic instructions. In this approach, the scheduling and non-determinism effects of the parallel process can also be handled. Each process in the KPN is annotated, control traces for a set of input data, and a symbolic program is generated. The architecture model contains a symbolic program unit (SPU) with read and write ports and FIFO buffers for communication. The SPU can process the application model generated by symbolic programs. The connection between the architecture resources is a point to point connection network. The SPU resources contain read, write, and execute units that can run in parallel. It also contains program and control units for scheduling traces from different processes according to their control and resource availability information. The SPUs are modeled in SystemC.

### **SESAME**

Similar to ARCHER, simulation of embedded system architectures for multi-level exploration (SESAME) is also based on the SPADE framework and uses a trace-driven simulation technique for system level performance evaluation. SESAME models the application using the Kahn process network either automatically using the Compaan framework [53] or manually from a sequential C/C++ code [54].

The architecture in SESAME is modeled in the Transaction Level, which simulates the traces' performance consequences carrying the application's computation and communication workload. The architecture is constructed from a generic library with configurable performance blocks for processing cores, memory, and interconnect subsystems. These generic blocks are modeled in Pearl [55] or SystemC and a SystemC-Pearl extension library [54].

Unlike ARCHER, where the scheduling (control) unit is integrated inside the processing element, SESAME has a separate mapping layer for scheduling the traces of processes mapped on the same architecture resource. This mapping layer consists of Virtual processing elements and FIFO buffers for communication between the virtual processors. The virtual processors read the event traces of the application model and dispatch them, according to different scheduling policies (FCFS, round-robin, or custom), to a processing element in the architecture model [54].

The structure of the process in the application model, their mapping, and the architecture model are described in YAML language, which can be rapidly changed for analyzing different design choices. Additionally, SESAME allows simulating the architecture at different levels of

abstraction. The architecture models can be gradually refined for more detailed performance analysis without modifying the application model. The mapping layer in SESAME refines the coarse-grained communication and computation event traces of the application into fine-grained computation and communication events using data-flow graphs to achieve detailed performance analysis.

## **TAPES**

Trace-based architecture performance evaluation with SystemC (TAPES) is a framework implemented in SystemC for performance evaluation of an SoC at the system level using the transaction level approach [56]. The application functionality and behavior are manually modeled as traces. Furthermore, traces are parsed and translated by the underlying architecture resources during the simulation. Each trace can include a sequence of an application task's execution primitives with some transaction primitives for interaction with other resources. The computation and communication primitives of the application trace are translated on the architecture resources, as latencies and transactions between the architecture modules, respectively. No real data is transferred between the modules; instead, tokens are transferred, and these tokens can trigger other traces in the destination module. The architecture resources are modeled in SystemC, interacting with each other by calling SystemC transactions via a single system-bus or point-to-point connections. The communication resources additionally model the dynamic system behavior such as resource contention and arbitration. TAPES is mainly focusing on network processor applications.

The control dependencies between the traces are resolved manually by generating traces for all possible execution patterns of the application. Moreover, system cache effects are also manually integrated into the traces before the start of the simulation. The system architecture is automatically generated from a model library containing abstract processors, memory blocks, accelerators, and communication resources. It is based on a system configuration provided by the user in an XML file. XML file enables easy evaluation of different mapping and application-architecture configurations. The traces and the pattern for the control path between the traces are manually specified within the traces.

## **MILAN**

Model based Integrated simuLAtioN (MILAN) integrates various simulators, with different level of abstractions and different input/output formats, into a single framework. Moreover, it provides a global user interface for system specification, enabling a system-wide performance evaluation of an embedded system design [57].

The hierarchical simulation technique in the MILAN framework has two levels. The top-level is based on the interpretive simulation tool, high-level performance Estimator (HiPerE), and the bottom level is based on component-specific simulators with different abstraction levels [58].

In [58], a system level performance evaluation technique based on the MILAN framework is used. The architecture is modeled using Generic models (GenM), which consists of generic



processors, reconfigurable logic, and memory blocks with some performance parameters such as timing and energy consumption. The application is modeled as a task graph, which shows the execution order of each task mapped either on the processor or reconfigurable logic resources. During the simulation, task preemption, dynamic scheduling, and other operating system effects are not considered. Each task's starting time is defined, considering the dependencies between the tasks and their estimated execution time.

HiPerE, starts from an initial set of component-specific performance parameters given by the user, estimates the overall system performance and generates a task activity report for energy consumption and latency estimates. Additionally, HiPerE generates the configuration file to configure the low-level stand-alone component-specific simulators of each architecture component. The low-level simulators' feedback is used back by the HiPerE to refine the system level component performance parameters for more accurate overall system performance results.

## **METROPOLIS**

METROPOLIS is an embedded design framework based on meta-models with precise semantics which can capture functionality, architecture description, the mapping relation between the application and architecture, and a logic language for capturing declarative and non-functional constraints of the design [59]. Since the meta-models have precise semantics, the METROPOLIS framework can support synthesis and formal analysis tools as well. Additionally, it provides some APIs that allow the designer to modify or add more information to the models.

The system application is modeled as a network of concurrent processes or tasks interacting with each other through ports with interface functions via a medium that implements these interfaces. Each process execution is a sequence of events that represent the entry and exit points of the code. The whole network's execution behavior is defined as a sequence of event vectors that trigger the processes during the execution. The meta-models can also capture non-deterministic behavior, and also logic formulas are used to define some constraints that coordinate the processes and behavior of the network. The architecture is also modeled as a network of processes, representing software tasks, processors, buses, and memory subsystems. The architecture model is event-driven, and it implements the functionality of the system by providing services to these events with costs, such as time or energy. The application model's processes request some services using events, and the architecture model provides these services by considering the performance cost. The mapping between the application and architecture model is established through a third network layer that synchronizes the events of processes in the application model and the corresponding resources' events in the architecture model.

In [60] a framework based on DIPLODOCUS [61] for system level architecture exploration, and performance evaluation of an embedded system focusing on the influence of shared resources on the performance of the system is presented. The application is modeled as a network of tasks whose behavior is modeled as a UML activity diagram, such as control flow commands and communication commands (sending sample data, waiting for events).

The tasks interact with each other through *channels* (for exchanging data samples), *events* (exchanging signals), and *requests* (requesting triggering of other tasks). Communication through *channels* and *events* are blocking or non-blocking while the communication through *requests* are non-blocking. The architecture is also modeled as a network of resources, instantiated from a generic library, for computation, communication, and storage purposes such as CPU, Memory block, buses, switches, and routers. The architecture resources' performance characteristics are configured through parameters. Following the Y-chart [62] approach, application tasks are mapped on the architecture resources, where each task can perform computation or communication requests to the corresponding architecture resource. To handle the resource allocation and scheduling of concurrent requests from different tasks mapped on the same resources, a virtual node (VN) component for each architecture resource is modeled. VN models an architecture resource's allocation and scheduling behavior, such as processors and bus, for concurrent requests from different application tasks. The allocation request considers the time for processing a pending request according to the underlying architecture resource's performance metrics. The framework takes the UML models of application, architecture, and mapping relation and generates SystemC code for simulation. The simulation generates statistics for the best, worse, and average execution time of each task and the utilization of each architecture resource.

In [63], the focus is on modeling and analyzing application using a task modeling language (TML). The application is modeled independent of the architecture, as a network of communication tasks. In addition to computation and communication primitives, control exchange between the task nodes, and a task flow control semantic is also considered in the task model. The tasks communicate with each other through virtual channels of finite or infinite capacity with specific data width. The exchanged information between the tasks is abstract-data and control-data. The control data consists of *REQ*(request) instructions and event notifications, which are used for task invocation and synchronization respectively. Each task's data processing functionality is abstracted by a sequence of three coarse-grained instructions *EXECl*, *EXECF*, and *EXECC* for fixed point integer operations, floating-point operation, and custom hardware instructions, respectively. These instructions' performance cost, such as execution time, depends on the underlying architecture, which is defined after mapping the application model on the architecture model.

## ARTS

Abstract system level modeling and simulation framework (ARTS) [64] is a SystemC-based framework for system level performance evaluation and modeling of an multiprocessors system on chip. ARTS mainly focuses on system level modeling and performance evaluation of streaming-based applications. In ARTS, application is modeled as a task graph, where nodes represent atomic application tasks and the edges represent the dependency between nodes. There are 3 types of task nodes, *computation*, *message*, and *I/O* tasks. Each task node has parameters, such as worst-case execution time (WCET), best-case execution time (BCET), release time, deadline, and memory requirement. Each task's behavior is implemented as a finite state machine with four states(*idle*, *ready*, *running*, *preempted*). The task's execution time

depends on the clock frequency and availability of the corresponding processing element in the architecture model to which the task is mapped. The architecture model comprises processing elements and communication elements such as buses, networks, or memory blocks. The processing elements are modeled as abstract RTOS models, which provide services, i.e., resource allocation, task scheduling, and execution synchronization. The architecture communication network is modeled as a communication-processor that can non-preemptively schedule the *message* type tasks of the application. A custom scripting language is used to model application, architecture and mapping their relation.

The other method for modeling and evaluating an embedded system's performance at a high level is using the specification and description language (SDL) [65]. In [66] the system specification and functionality are modeled in SDL, and the architecture and mapping of application tasks on the architecture resources are modeled by annotating the SDL code processes with *NODE*, *PRIORITY*, and *DELAY* directives. Where *NODE* represents the architecture resources such as processors or dedicated hardware, *PRIORITY* represents the priority of each process, and *DELAY* represents the execution time of each task on the corresponding architecture resource. These architecture performance-related parameters, such as execution delay, are obtained by generating and simulating a low-level implementation of each SDL application process as hardware (VHDL RTL code) and software (C code). The annotated SDL code is simulated in the GEODISIM system level simulator to evaluate different hardware software partitionings and architecture configurations. Each SDL process to evaluate an application model's performance for different architecture models with different mapping relations is implemented as hardware (VHDL RTL code) and software (C code).

### 2.2.2 ESL Power Estimation Frameworks

Electronics system level simulation frameworks that rely on functional simulation to provide the embedded system's timing and functionality are extensively studied in academia and industry. Works from academia and industry relevant to this dissertation's scope are discussed in this section and categorized based on two practical approaches. First, those electronics system level frameworks that connect the power model to the simulation framework and second, those frameworks that integrate the power modeling information within the functional simulation.

#### ESL Frameworks Integrating Power Models into SystemC Simulation

[67] integrates *power hooks* into SystemC simulation instead of integrating the complete power models. A proposed software infrastructure integrates the *power hooks* into SystemC simulation, and a timed functional model calls them. The number of *power hooks* and types required depends on the components. [67] claims system-wide rapid power estimation and analysis.

An approach called *pktool* is presented in [68]. This tool creates power modules as a wrapper on top of SystemC modules. Furthermore, power modules observe the data relevant

for power models from the input and output ports. This work intends to keep the SystemC simulation intact as much as possible.

A work tightly coupled with SystemC is presented in [69] as it modifies the base class for modules with member function to add energy and power types to set dynamic and static power consumption. Power consumption comes from the components' behavioral code in the embedded system modeled in SystemC/TLM. The limitation of [69] results from replacing the generic TLM transaction payload structure with a customized payload structure for power annotation by observing the switching bits.

An electronics system level framework for power and temperature modeling in SystemC was first discussed in [70]. It spreads the power peaks over the correct time interval in order to get accurate thermal simulations. This work was further enhanced in [71, 72] by connecting the power model developed in external power/temperature solver to SystemC/M functional simulation. The work done for the SystemC extension is available as an open-source library called *LIBTLMPWT* [73].

### **ESL Frameworks Connecting Power models to SystemC Simulation**

A power estimation framework built on top of the architecture exploration framework SESAM [54] is introduced in [74]. In this framework, the application model is abstracted and modeled by the abstracted Kahn Process Network [75]. The simulation events are transferred to the hardware model, and the hardware model maps it to the event tables, which contain the time and power budget. The approach used for application modeling is very similar to the approach used in this dissertation.

A project named *COMPLEX* [76] estimates power in the context of design space exploration and rapid prototyping of embedded systems for hardware-software co-design. Power models are developed using the PSM approach, but with a slight variation, the *protocol state machine* is placed between SystemC simulation and power models. The *protocol state machine* observes the transactions of functional simulation and determines the power states in PSM, which helps black-box component modeling [77, 78]. This work separates the application from the platform and the functional simulation from non-functional properties, i.e., the power, which is also the focus of this dissertation.

A research group has worked towards multiple power estimation frameworks [79, 80, 46]. Each framework has a different methodology and the logic behind calling it hybrid.

The work [79] has a similar approach as discussed in [76]. This approach is called a hybrid approach as it can model both white and black boxes. Instead of a protocol state machine as in [76], it has component estimator modules attached to the ports of the components' functional models. The work in [80] uses low-level physical measurements and analytical formulas to model the power of vital components in an embedded system and use the electronics system level simulation counters to multiply them with the parameters determined by physical measurements to calculate the power consumption. Electronics system level simulation counters are put for those events that impact power consumption. This research group's last work [46] is also a hybrid approach, as a functional power estimator is connected with fast electronics system level simulation at the transaction level. Electronics system level simulation

(modeled at transaction level) drives different power models of the components, and the power models are developed by linear regression in the first place.

### System Level Performance and Power Evaluation in Industry

*Intel CoFluent Studio* [81], as the name suggests, is an EDA tool powered by Intel. It is a commercial model-driven tool for designing and exploring complex embedded systems in the early phases of the design process. In CoFluent, the application is modeled, using a GUI tool, as a network of function blocks or processes communicating with each other through communication elements such as events, shared variables, or message queues. Activity diagrams with timing attributes abstract the complex algorithms or behavior of each process. Once the application is modeled, a timed SystemC TLM model of the application can be automatically generated. The simulation of this timed model can be used for the analysis of the application model. Similarly, using the same GUI tool, the system architecture model is created by connecting different generic architecture resources, such as processors (ASIC, CPU, MCU, DSP, FPGA), interrupts, interconnect(BUS), and memory blocks. High-level performance attributes characterize architecture resources. The mapping relation of the application model with the architecture model is expressed by creating mapping models of the system. In the mapping models, each function or process in the application model is mapped on one of the architecture models' processing resources. Similarly, the communication path between the application model processes is mapped on the interconnect subsystem in the architecture model. The shared variables and message queues in the application model are mapped on the architecture memory blocks. The SystemC TLM code of the system can be generated automatically to analyze the power, cost, resources loads, memory footprint, and dynamic behavior of the application on the architecture.

*Synopsys Platform Architect (Synopsys PA) MCO* is a SystemC TLM standard-based graphical framework for system level design and early power, and performance analysis of a multicore SoC architecture [16]. In Synopsys PA, the application is modeled as a task-graph. Each node is instantiated from basic blocks of a generic task library (GTL) and represents an application task in a task graph. The edges in the task-graph represent the nodes' dependencies, and this edge connection between two task nodes is established via ports. The task nodes use samples of communication tokens for activation and synchronization purposes. Tasks read their input ports, consume or process the input tokens and generate output tokens. The behavior and execution time of each task for actual processing of the incoming tokens is specified by some generic functions, such as for data processing (*cpu function*) and memory access (*mem function*), which are connected to each task node during instantiation of the corresponding node. Under the control of a default task manager, the application model can be simulated stand-alone to analyze the application behavior and memory traces. The architecture model components are instantiated from a library containing a virtual processing unit (VPU), memory blocks, and interconnect subsystems. The VPU, when a task with *cpu function* is mapped on it, can generate statistical traffic for instruction(*fetch*) and data (*load + store*) accesses. The mapped task graph with architecture can be simulated to analyze the system's overall power and performance.

## 2.3 Fundamentals

After explaining the background and related work, a methodology's fundamentals with a defined goal of power and performance estimation, co-optimization, and design space exploration to analyze different variants and configurations of the embedded systems' heterogeneous architecture are presented in this section.

An embedded system design flow, which starts from design specifications by the customer, has multiple stages until completion. In this dissertation, the complete design flow is divided and modeled into three different abstraction levels.

1. Architectural abstraction level
2. Software abstraction level
3. Hardware abstraction level

Before explaining these abstraction levels from a methodology point of view and comparing these with the Y-chart [7], it is vital to understand the essential components of a methodology at these abstraction levels.

**Application model** is a model representing a use case, a scenario, the activation flow, task sequence, or order of processes depending upon the systems' abstraction level and the information available. Software that is developed from system specifications, if available, provides an accurate representation of an application model. However, most of the time, the software is not present from the start of the design flow.

**Architecture model or performance model** is a model capable of simulating a system level use case, application, or existing software to generate parameters for performance and power analysis of the system. It consists of fundamental components like processors, programmable and dedicated hardware accelerators, memory, and buses, modeled at a suitable abstraction level.

For early system level evaluations, these resources in the architecture model capture only the application's execution behavior and provide means to measure the system's performance details. These details include high-level throughput, latencies, deadlines, and bandwidth analysis. Therefore, they are also called performance models.

**Power model** is a model containing power modeling information of the embedded system's components contributing significantly towards power consumption, i.e., core, on-chip and off-chip memories, buses, hardware accelerators, clock tree. The power model is driven by the architecture or performance model's parameters to perform a static or dynamic power analysis. They can also carry out a standalone power analysis of the modeled system, i.e., with power state machines (PSM) or handwritten scenarios.

Ideally, the power model should be developed at the same granularity as the architectural model. Nevertheless, the system use cases relevant for performance and power analysis are not always the same; hence, the power and architecture model's granularity are not always the same.

**Automation** is the integration of the application model, architecture model, and power model in a methodology capable of power and performance analysis and co-optimization requires automatic interaction of the models. This automation can be achieved with the help of scripting, application programming interface (APIs), value change dump (VCD) file, java script object notation (JSON) file depending upon the requirements and abstraction level.

Now, we explain the three abstraction levels of the methodology in the context of essential components introduced above.

### 2.3.1 Architectural Abstraction Level

The high-level definition of architectural abstraction in the context of this dissertation is a level that is pre-software and pre-silicon. Thus, neither software nor hardware is available for an embedded system. It corresponds to the electronics system level discussed in detail in section 2.1.1.

The architectural level is the first abstraction level of the embedded system, and it is the most important one for design space exploration and key performance indicators analysis. According to the Pareto principle [82], the correct design decisions taken at an early phase of the project have (80%) or significant impact than the later stages' decisions (only 20% impact).

For example, hardware-software split, components placement on a silicon die, selection of the technology node, size of the memory, an on-chip of-chip split of the applications are few examples of the decisions system engineers have to take to translate design specifications into system configuration. A methodology at the architectural abstraction level that can address system engineers' need to evaluate different system configurations for system level use cases is particularly advantageous - as it can help refine the application and architecture model of the embedded system.

At this abstraction level, the challenge to develop a methodology is the missing low-level implementation details. Application models are abstracted, such as directed acyclic graphs (DAG), control data flow graphs (CDFG), finite state machines (FSM), kahn process networks (KPN), or task graphs. Architecture models are developed with the help of language capable of abstracting hardware functionality. Transaction-level modeling (TLM) libraries on top of SystemC is an attractive choice to model architectural models - as they separate implementation details of computation from communication. Moreover, they can be enhanced to simulate the application model to generate system parameters responsible for performance and power analysis. At this stage, power models consist of an excel file filled with the information coming from data sheets, designers' experience, or rule of thumbs practiced in the industry.

The higher the abstraction, the lower the accuracy - as low-level implementation details are missing, and many assumptions are made along with methodology development. An advantage of higher abstraction can be seen in the result of fast simulation speed; otherwise, simulating one millisecond of a cycle-accurate system can take more than hours, which is not practical to simulate system level use cases.

In conclusion, a methodology that can analyze and co-optimize power and performance at the architectural abstraction level is desirable but challenging to develop. Hence, this work

focuses on the architectural abstraction level and linking it with lower abstraction levels.

### 2.3.2 Software Abstraction Level

The software abstraction level in this dissertation corresponds to a level that is pre-silicon but post-software. Similar to the architectural abstraction level, the hardware is in the development phase, but the software engineers have developed the software at this stage of the project.

The architecture/performance model development of the essential components discussed in Section 2.3 to achieve a methodology at the software abstraction level has the following potential options.

Application and key performance indicator critical system level use cases are modeled with the existing software; hence, the application model's accuracy is very high. The second vital component of the methodology is the architectural/performance model. Similar to the architectural abstraction level, hardware components can be represented with SystemC/TLM modules (that abstract computation and focus on communication) at the Software abstraction level. virtual prototypes (VP) composed of SystemC/TLM modules and libraries representing the architecture model (or embedded system platform) can simulate a complete software. SystemC/TLM VP modeling has different time notions loosely-timed (LT) or approximately-timed (AT), and there can be a trade-off for simulation speed or accuracy. This abstraction level is also known as the functional verification stage in embedded system design. The software and firmware are both at hand, and the system level use case or activation flow of the actual hardware can be triggered with the help of test-cases to verify the functionality. Functional verification gives confidence that the system is performing as expected.

Power models are refined during the architectural abstraction level. They are developed in electronic design automation (EDA) tools or frameworks to analyze fully-grown software at this abstraction level. Power models developed in EDA tools or frameworks have multiple options to simulate a performance/architecture model's stimuli.

The power model can be attached to the software in the form of libraries, or an XML file is read at SystemC elaboration phase or with software APIs to external power modeling tools. For example, virtual prototypes of the architecture model can simulate an application or system level use case and dump the performance and power-related stimuli. The dump file in the form of value change dump or another format can be re-used by an external power modeling tool or framework to produce a dynamic power consumption trace over time.

The software abstraction level accuracy in terms of an application model is very high as low-level implementation details related to an application or system level use case are known. The simulation of the architecture/performance model depends upon the trade-off made between accuracy and speed.

Depending on the system scenario under consideration, different timing resolutions and scales are needed. Mobile communication systems have characteristic time constants (e.g., transmission time interval (TTI) of 1 ms in 4G). For functional simulation of a 4G use case, a performance/architecture model needs to simulate several hundred TTIs. Therefore, a performance model development at a TLM loosely-timed level is a useful abstraction.



For throughput and latency analysis of the memory subsystems and interconnects, timing accuracy at a transaction level is needed and requires a TLM approximately-timed (AT) modeling style.

### 2.3.3 Hardware Abstraction Level

The third and last abstraction level in this dissertation is the hardware abstraction level. It is a post-silicon, and post-software abstraction level, unless if the embedded system developed is generic enough to support a new application model or the new hardware is a slight modification of the legacy hardware. In that case, this abstraction level is post-silicon but pre-software.

There are certain advantages of this abstraction level; for example, the application model's simulation or running an end to end system level use case is easy and fast. The regression-based power modeling techniques require actual hardware to run at least one system level use case to develop a power model at this abstraction level.

Post-silicon and post-software is the lowest abstraction level. Thus, low-level implementation details are available, which allows for developing a detailed and precise design methodology for embedded systems. For accurate and precise results, modeling can be done at a fine granularity, which requires higher modeling efforts from model developers. So, accurate and precise results are achieved at the cost of higher modeling efforts.

Post-silicon implies that the embedded system's hardware is constant in the hardware abstraction level; thus, the room to perform design space exploration and optimize power and performance key performance indicator is limited. As with given hardware, the flexibility to choose between a different type of core, memory type, on-chip off-chip memory size, technology node, placement of the component, and bus sizes is limited.

### 2.3.4 Holistic and Modular Methodology

The holistic and modular methodology in the context of this dissertation is defined as a methodology that starts from the architecture, incorporates software and hardware abstraction level for design space exploration, power and performance estimation and analysis. Moreover, still modular enough to keep the critical components of methodology, i.e., application, architectural, and power model, separate.

In real projects, these different parts are developed by different teams. Hardware architects define different variants of the overall architecture. System architects define the high-level structure of the system use cases (functional elements and their interaction). Power experts do the early power analysis, defining early abstractions of power models at the component level, derived from previous generations or high-level estimates, and including all relevant technology parameters.

The main contribution of this work is a *Power and Performance Optimization and Exploration Methodology* POEM to address this a distributed project setup by *orthogonalization of concerns* [6].

The key to holistic methodology is power modeling framework, which is integrateable with multiple abstraction levels and acts as a binding force between them, as shown in Figure 2.3.

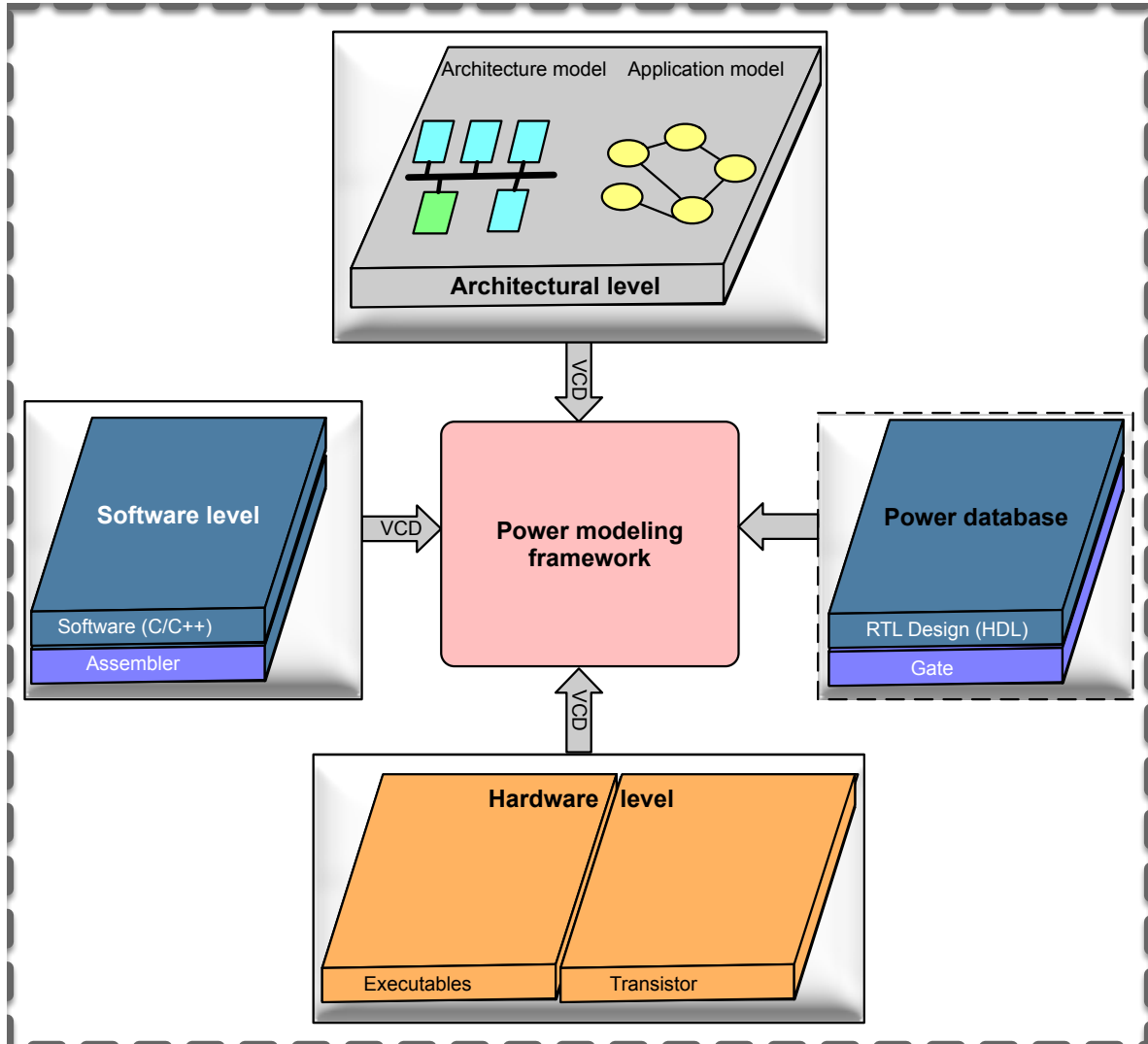


Figure 2.3: POEM: A multi-abstraction level methodology, estimating KPIs at architectural, software, and hardware abstraction level with a common power modeling framework. That implies a methodology in which the modules addressing functional simulation, power modeling, or application development are agnostic concerning each other.

## 3 POEM: Power and Performance Optimization and Exploration Methodology

After stating the background, fundamentals and terminologies in Chapter 2, this chapter comprises the details of application, architecture, and power model. Application and architecture models are introduced at the architectural abstraction level (2.3.1) because at this level, simulation platforms and software are not straightforward to model due to missing implementation details.

The application model is represented with the help of task graphs. The performance model simulates the task graph and runs the application model to generate performance parameters and report the power evaluation stimuli. Both application and performance models are developed with SystemC/TLM and an additional library on top. As this work is an outcome of the collaboration with the Intel Cellular department, the application and performance models used in this dissertation are developed internally by intel using intel SystemC transaction-level modeling (ISCTLM).

ISCTLM is a wrapper that provides instrumentation for the parameters' configuration of task graph and performance model parameters to evaluate a different set of application and architecture model combinations. It allows parameters' configuration dynamically - that means not only an application and a performance model can be modified by just modifying the configuration file, but also the parameters impacting the power and performance can be changed at run time. For example, frequency and voltage change during a simulation run while switching from one use case to another or moving from short connected mode discontinuous reception (CDRX) to fully connected mode in a cellular modem. This ISCTLM library is fully compliant with the Accellera [83] (SystemC based) library.

An in-house Intel framework exercising ISCTLM for performance and application model simulation is called system architecture and validation environment (SAVE). One of power and performance optimization and exploration methodology (POEM) goals is modular setup; hence, this dissertation uses the SAVE environment to analyze power and performance by attaching an agnostic power model to it and making this setup in line with our goal.

Sections 3.1, 3.2 explaining the application and architecture models have been presented in [84] and in [85].

### 3.1 Application Modeling

In our methodology, given a set of system specifications, an application is manually modeled as a task graph. Nodes represent abstract application tasks, and the edges represent the

dependencies and the communication channels between the nodes. These communication channels are modeled as bounded SystemC first-in-first-out (FIFO) channels. The user defines the size of each FIFO channel. During the simulation, the nodes are triggered when there is data available in their input FIFO channels. The triggered nodes first read their input ports, request executing on their mapped architecture resource, and write some communication-data to their output ports to trigger other tasks.

The task graph captures the functionality of the application without implementing the actual behavior of the application tasks. Each application task's actual data processing or behavior is abstracted to a set of primitives representing the execution workload of a task on an architecture resource, from a performance point of view. These workloads approximate the computation and communication activities as if it was running on a real platform. The task graph provides the underlying architecture resources' simulation models with each task's execution workload in an application or use case while maintaining the correct control and data flow of the application. Consequently, the behavior of the overall architecture for the corresponding use case or application can be analyzed.

A task node's characteristics are defined by a set of parameters, such as start or release time, deadline, a period, and priority. Moreover, each node has a parameter for referencing a resource in the architecture model. Before explaining the application stand-alone simulation and mapping of the application task on the architecture resources for co-simulation of the application-architecture model combination, the following section presents the concept of abstract task execution workload or task-execution model in more details.

### 3.1.1 Task Execution Model (Workload)

The goal is to model the application generically, such that it is independent of the architecture model. We want to compare the application task's behavior mapped to different architectural resources for power and performance analysis and comparison. Therefore, the actual behavior of processing data is not implemented inside the tasks in the task graph. Instead, each task only provides its processing workload, corresponding to input data sets, to drive the underlying architecture performance models. A set of parameters captures this execution or data processing workload of a task on an architecture-resource.

For example, suppose one of the application tasks, which is mapped on a DMA module, is to transfer a specific volume of data from external memory to on-chip memory. Furthermore, suppose the DMA performance model is very abstract and has only three run-time parameters, source address, the volume of data, and destination address. Then the very abstract execution workload of the task mapped on the DMA module is the address of the external memory, the volume of data to be transferred, and the address of the on-chip memory block. Whenever this task is triggered, it will request executing on the DMA module by providing a reference to its execution workload. The DMA module's run-time parameters will be configured according to the values provided by the task's execution workload. Accordingly, the DMA will initiate a read transaction request on the interconnect for reading data from the corresponding source address(external memory), and it will be followed by a write transaction request to the destination address (on-chip memory). This task's completion time depends on the load,

bandwidth, latency of the interconnect, the volume of data, and the read and write access latency of the external and on-chip memory blocks. Depending on how detailed the DMA and the memory subsystems are modeled, this task execution workload parameters can be refined further for including the details about reading, writing, and preparing each chunk of data during this transfer.

In our methodology, for each task, we encapsulate this set of execution parameters as member variables into a SystemC module, which we refer to as workload objects (`workload_obj`). The reason why the SystemC module is used instead of plain C++ struct containers is the ease in initializing these member variables from a configuration file, which is discussed in detail in Section 3.1.2. These workload objects represent coarse-grained computation and communication activities of a task’s execution on an architecture resource during a simulation. The parameters for these activities depend on the type and modeling granularity of the mapped architecture resource. For instance, when a task is mapped on a processing resource modeling the performance of a CPU with different levels of caches, then it is necessary to have some parameters in the task’s workload object to represent these cache effects. However, if the same task is mapped on an application-specific integrated circuit (ASIC), without caches, then the workload object will not need these cache effect parameters.

Moreover, the number and granularity of these parameters or activities within a workload object depend on how detailed the corresponding architecture resource, which the task is mapped on, is modeled. These workload objects are instantiated as sub-modules within the architecture resources. In the task graph level, each task only carries a reference or name of these objects, without considering the internal parameter details of them. Therefore, this enables the application to be modeled independently of the implementation details of the architecture resources.

Table 3.1 shows the `workload_obj` parameter of a task mapped on a virtual engine. Virtual engines, which will be discussed with details in Section 3.2.2, do not generate any traffic while executing a task; instead, they only simulate the execution latency of the tasks mapped on to them. Therefore the workload of each task mapped on a virtual-engine is only the execution time. This is captured by a single parameter, `exe_time`. Similarly, the workload of a task mapped on a more detailed DMA module is shown in Table 3.2. Besides the source, destination, and volume of data, it also has parameters for dividing this data volume into bursts and chunks. Accordingly, traffic is generated on the interconnect for each burst transfer, and the corresponding memory blocks are accessed.

Table 3.1: A task execution workload (`workload_obj`) mapped on a Virtual Engine

Parameter Name	Data Type	Default Value	Details
<code>exe_time</code>	<code>sc_time</code>	0 ns	Task execution time

Table 3.3 shows a task execution workload for a task mapped on a typical stochastic CPU. The total execution of a task on a CPU might need a specific number of instructions. This is defined by the `numberOfInst` parameter. The CPU’s internal behavior will loop for the total number of `numberOfInst` times, where each iteration means executing an

Table 3.2: A typical task execution workload (`workload_obj`) mapped on a DMA channel

Parameter Name	Data Type	Default Value	Description
SMEM_Addr	sc_dt::uint64	0x0	Source Memory Address
DMEM_Addr	sc_dt::uint64	0x0	Destination Memory Address
TRBKsize	unsigned int.	0	Transfer Block: total volume of data to be transferred
BRSTsize	unsigned int.	0	Bytes per Burst transfer
TRBKcyc	unsigned int.	0	Number of cycles required for preparing a transfer block of data
BRSTcyc	unsigned int.	0	Number of cycles required for preparing a burst transfer transaction
CHNKsize	unsigned int.	0	Chunk: bytes per codeblock or codeword, which depends on the width of the interconnect
SMEM_Size	sc_dt::uint64	0x0	Source Memory Size
DMEM_Size	sc_dt::uint64	0x0	Destination Memory Size
SMEM_CHNK_Base	sc_dt::uint64	0x0	Source Memory Address for first CHNK in case of fixed start
DMEM_CHNK_Base	sc_dt::uint64	0x0	Destination Memory Address for first CHNK

Table 3.3: A typical task execution workload (`workload_obj`) mapped on a CPU with instruction and data caches

Parameter Name	Data Type	Default Value	Description
<code>numberOfInst</code>	Integer	0	Total number of instructions for a task
<code>fetchRatio</code>	double	0	Probability that a task execution performs a fetch operation
<code>fetchHitRate</code>	double	0	Probability that a fetch operation is a hit
<code>loadRatio</code>	double	0	Probability that a task execution performs a load operation
<code>loadHitRate</code>	double	0	Probability that a load operation is a hit
<code>storeRatio</code>	double	0	Probability that a task execution performs a store operation
<code>storeHitRate</code>	double	0	Probability that a store operation is a hit
<code>lineDirtyRate</code>	double	0	Probability that a cache line is dirty
<code>branchRatio</code>	double	0	Probability of branching
<code>branchMispredict</code>	double	0	Probability of branch miss prediction
<code>instAddOffset</code>	<code>sc_dt::uint64</code>	0x0	Task inst. address offset to the inst. mem. base address
<code>dataAddOffset</code>	<code>sc_dt::uint64</code>	0x0	Task data address offset to the data mem. base address

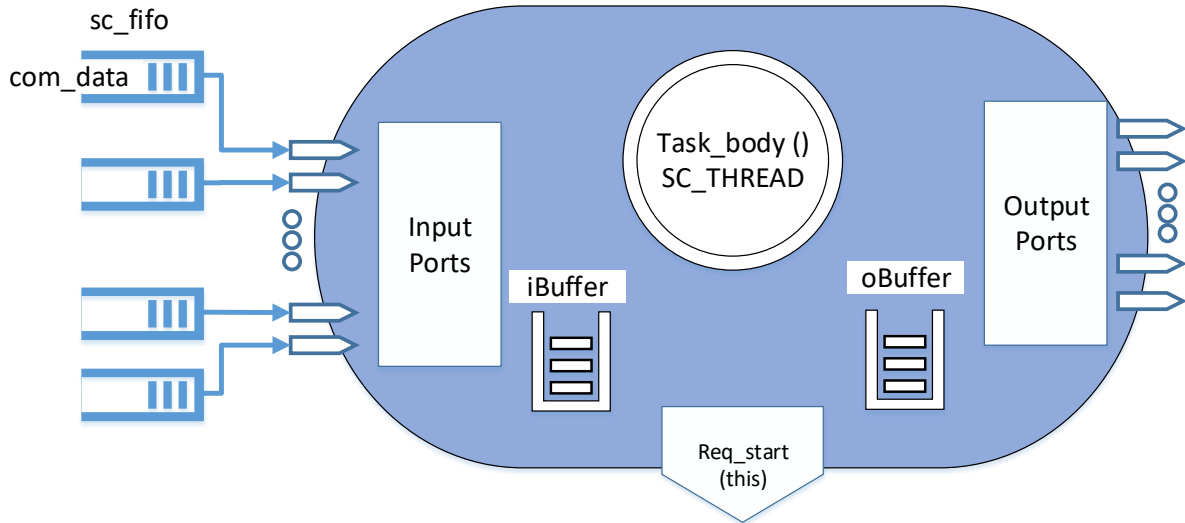


Figure 3.1: Node of a task graph: Input is the abstract communication data coming from SC\_fifos connected to the input ports and finally ending up in input buffers. The method/function SC\_THREAD controls the node behavior and processes the data to the output buffer. The output buffers transfer data to the output FIFO channels. According to the provided mapping information, the Req\_start member of the task graph node requests execution of the task on an underlying architecture.

instruction. During the execution of an instruction, fetch, load, store, and execute operations are performed, and accordingly, memory blocks are accessed, and traffic is generated on the interconnect. The *miss* or *hit* for any data or instruction fetch operation is decided during run-time according to the probabilities given in Table 3.3. The penalty for accessing different memory blocks will depend on the traffic load of the corresponding memory block's interconnect and performance specifications.

### 3.1.2 Implementation of the Task-Graph in SystemC

#### Task-Graph Nodes

The task graph nodes have a set of input ports for reading abstract communication data from their input FIFO channels, and output ports for writing to their output FIFO channels, as shown in Figure 3.1. The number of input and output ports of a task is defined by the number of its predecessor and successor tasks, respectively. The communication data between the nodes is a generic C++ container, and a task is triggered when it receives a sample of this communication data in its input ports. By default, the communication-data is only used for triggering the execution of tasks. However, in addition to triggering information, it can also include data such as global variables for synchronization of tasks and control data for changing the internal control behavior of a task.

As shown in Figure 3.2, there are four kinds of nodes in our task graph methodology.



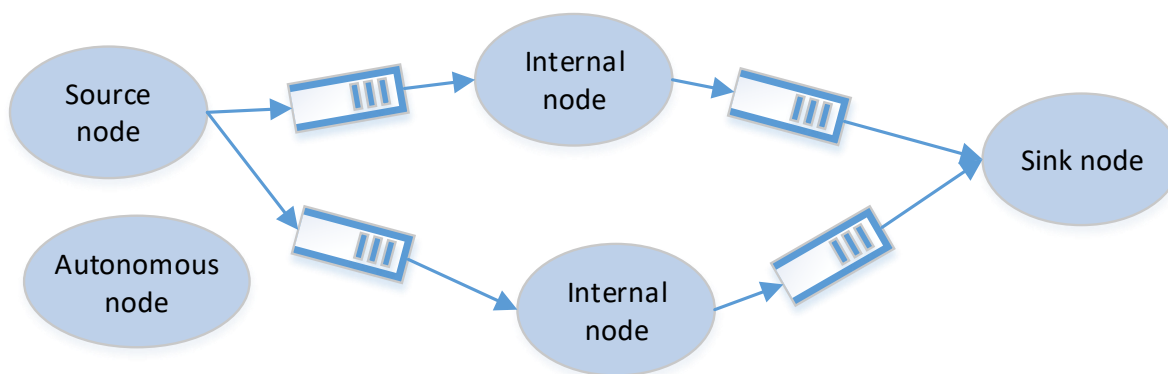


Figure 3.2: Illustration of source, sink, internal and autonomous nodes. FIFO buffers (communication channels) between these show the relation between input and output ports.

- 1) *Source node*: It has one input port and at least one output port. Since it has no predecessor tasks, the input port is left open. But in order to be triggered at least once during simulation, a default sample of communication-data is pushed to its input port during the instantiation of the task graph.
- 2) *Sink node*: It has at least one input port, connected to its predecessor tasks, and no output ports.
- 3) *Autonomous node*: It has only one input port and no output ports. This task has no connection or dependencies on the rest of the task graph. Like a source node, a default sample of communication-data is pushed to its input port during the task graph instantiation.
- 4) *Internal node*: This is any node within the task graph with at least one input and one output port connected to its predecessor and successor tasks, respectively.

A task graph node is a SystemC module, and its internal behavior is implemented as a SystemC thread (`SC_THREAD`). Each node has a set of parameters, method (function), and SystemC container members. These members are used to characterize the behavior of the task and establish its connection with other nodes during the task graph construction. Table 3.4 shows some of the main parameters of a node with a short description of each. Besides these parameters, each node has a set of methods (functions) that implement the node's internal control behavior and its interaction with other nodes. Table 3.5 shows these methods with a short description.

Most of these parameters are defined as `Attribute<T>` objects of ISCTLM. This allows us to configure and initialize these parameters for each task through a configuration file, as shown in Figure 3.3. This will be elaborated with details in Section 3.1.2.

During a simulation, each node can be in one of the five states *IDLE*, *PENDING*, *ACTIVE*, *PREEMPTED*, and *COMPLETED*. The execution flow of all four types of the task graph nodes

Table 3.4: General parameters of a task graph node

Parameter Name	Data Type	Default Value	Description
task_name	string	"?"	Name of the task
state	enum	IDLE	States of the task, <i>IDLE</i> , <i>PENDING</i> , <i>ACTIVE</i> , <i>PRE-EMPTED</i> , <i>COMPLETED</i>
task_id	integer	0	Unique ID of the task (defined during instantiation of the corresponding task graph node)
engine_name	string	"?"	Name of the engine which this task is mapped on
engineif	reference	NULL	A pointer to the task scheduler interface of the mapped engine
start_time	sc_time	0 ns	Task's release time
period	sc_time	0 ns	Task period (only valid for periodic tasks)
tot_num_itr	integer	0	Total number of iterations (only valid for periodic tasks)
trig_after_itr	integer	0	Total number of executions or iterations of a task before triggering its successor tasks
deadline	sc_time	0 ns	Deadline of the task
priority	integer	0	Priority of the task
workload_obj	string	"-"	Name of the execution workload of the task
workload_obj_ptr	reference	NULL	Pointer to the workload_obj
sel_prob	double	1.0	Execution probability of the task during simulation
suc_tasks	string	"-"	Name of the successor tasks
oTrig_mode	enum	OR	Output-ports triggering mode of the task
iTrig_mode	enum	AND	Input-ports reading mode of the task

```

32 HARDWARE.use_VEngines           = true
33 HARDWARE.TG_Constructor.generate_vcd   = true
34
35 HARDWARE.TG_Constructor.task1.task_name = task1
36 HARDWARE.TG_Constructor.task1.engine_name = HARDWARE.TG_Constructor.Vir_Eng
37 HARDWARE.TG_Constructor.task1.priority = 0
38 HARDWARE.TG_Constructor.task1.period = 10 us
39 HARDWARE.TG_Constructor.task1.start_time = 2 us
40 HARDWARE.TG_Constructor.task1.end_time = 0 us
41 HARDWARE.TG_Constructor.task1.tot_num_itr = 1
42 HARDWARE.TG_Constructor.task1.num_itr_trig = 1
43 HARDWARE.TG_Constructor.task1.out_trig_mode = AND
44 HARDWARE.TG_Constructor.task1.in_trig_mode = AND
45 HARDWARE.TG_Constructor.task1.suc_tasks = task3 task4 task7
46 HARDWARE.TG_Constructor.task1.wokload_obj = task1_workload
47
48

```

Figure 3.3: A piece of configuration file depicting a task’s specification using all or a subset of task graph node parameters.

Table 3.5: Main function members of a task graph node

Function	Description
void configure_node (sc_trace_file *tf)	Initializes some parameters and local variables according to the configuration file and also registers node’s state as a signal to a common trace file.
void new_input (sc_fifo*, node*)	For every successor and predecessor tasks of a node, these functions are called for instantiating new input and output ports and establishing their connections with the corresponding FIFO channels, during the construction of the task graph
void new_output (sc_fifo*, node*)	
void task_body()	This function is registered as SC_THREAD in the SystemC kernel. It implements the main behavior of a node.
void read_iports()	Reads the input ports according to AND or OR rules
com_data read_iData()	Reads received communication-data samples from input buffer of a node
void req_start()	Requests execution of the task on an underlying architecture according to the provided mapping information.
void write_oData(com_data)	It writes communication-data samples into the output buffer (oBuffer).
void trigger_out (trig_mode mode)	It triggers successor tasks by writing communication-data samples from the output buffer to the corresponding output ports of the node according to AND or OR rules.
void report_()	It writes the run-time statistics of the task into an activity report for performance analysis.

(source, sink, autonomous, internal) is shown in Figure 3.4. For the sake of clarity, we divide this flow, which represents the task's main thread (`task_body()`), into the following four steps:

Step-1) *Wait for starting time*: Each task waits for a specific time before starting execution, which represents its release time. This waiting time is defined as a parameter of the task and initialized from the task graph configuration file. At this step the node is in *IDLE* state.

Step-2) *Reading the input ports*: If the input data buffer is empty, then function `read_iports()` is called from the main thread (`task_body()`) of the task. If the input buffer is not empty, then the task continues with Step-3. The `read_iports()` function reads data of the input FIFO channels via the input ports of the node and pushes the received communication-data samples into the input buffer of the node. The reading from the channels is blocking, and it is performed according to the two "input-port reading modes" (*AND* or *OR*) of the task, which is defined by configuring the `iTrig_mode` parameter of the task. These modes, for instance, can represent a logical combination of a set of conditions in an *IF* statement of application code. For each mode, the execution of the node proceeds as follow:

- *AND mode*: In this mode, the task is triggered if all of its input FIFO channels have data. `read_iports()` function, reads all the input ports in a round-robin fashion. If any of the input FIFO channels is empty, it means that not all of this task's predecessors are completed. The task waits until all the running predecessors are completed, and communication-data is available on the corresponding FIFO channels.
- *OR mode*: In this mode, the task is triggered if any of its input FIFO channels have data. Since, reading from the input port is blocking, `read_iports()` function first checks the input FIFO channels for existence of data. If data exists then, it is read via the corresponding input port. In case all the input FIFO channels are empty, then the task waits for a `data_written` event, which will be notified by any input channel upon receiving a communication-data. The received data is read, and the task proceeds with the next steps.

Additionally, user can override `read_iports()` for implementing a custom method of reading the input ports.

Step-3) *Processing data*: In the data processing step of the task execution flow, shown in Figure 3.4, a task reads its input buffer one by one and for each received communication data, requests starting its processing on a resource by calling `req_start()` function of the node. By default, the `req_start()` function, calls the `req_start_task(this)` API in the task scheduler interface of the corresponding mapped processing element in the architecture model. This task's state is changed to *PENDING*, and the scheduler schedules its execution if the processing resource is busy with a previous task having

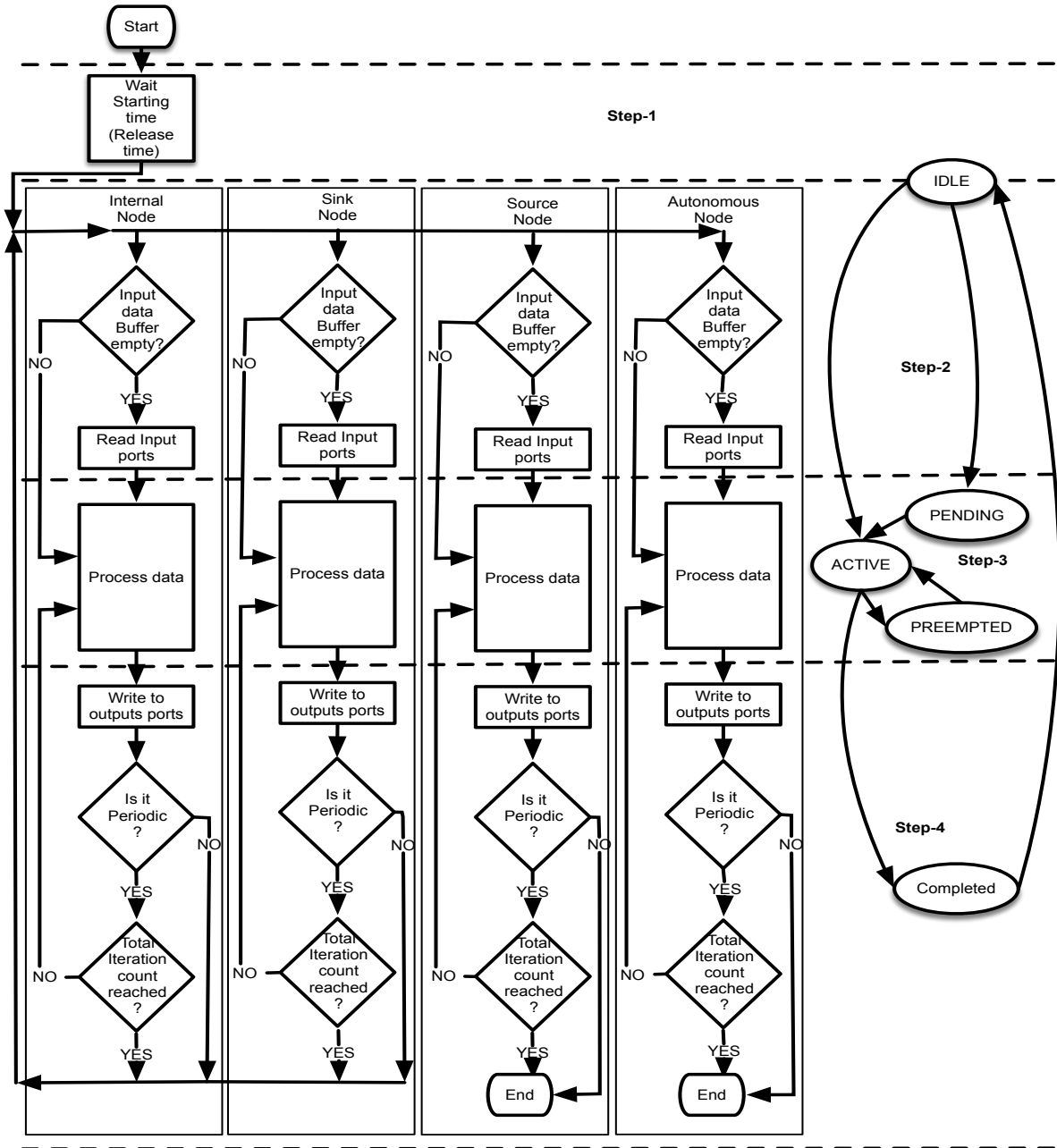


Figure 3.4: During the first step, the node waits for the release time of the task to expire, and in the second step, it waits for input ports to read the communication data. The third step is the core execution state of the execution flow, and the task graph node can take *ACTIVE*, *PREEMPTED*, or *PENDING* states depending upon the situation. Fourth is the final step, and the task graph node finishes its job and puts execution flow into the *COMPLETED* state.

higher priority. The previous task is preempted, and this new task is executed with its state changed to *ACTIVE* if the previous task has lower priority and is preemptible. As shown in Figure 3.4, in the processing step, a task can be in one of the three states, *PENDING*, *ACTIVE* and *PREEMPTED*. The completion time of a task depends on the load of the processing element, scheduling and resource allocation policies of the scheduler interface, and the previous tasks' execution time. The user can override the behavior of the `req_start()` function, in case some pre-processing of the received communication-data is necessary before requesting execution on an architecture resource.

Step-4) *Writing to output ports*: When the processing of a received data is completed, the task triggers its successor tasks using `trigger_out()` function. The `trigger_out()` function writes communication data samples to output FIFO channels via the output ports of the node. Writing to output ports, similar to reading input ports, is performed according to "output triggering mode" (*AND* or *OR*) of the task. This is defined by the `oTrig_mode` parameter of the task.

- *AND mode*: In this mode, the task triggers all of its successor tasks. Therefore, `trigger_out()` function writes same data samples to all output ports of the node in a round-robin fashion.
- *OR mode*: In this mode, the task triggers only one of its successor tasks. Each task has a probability of execution, which is defined by the `sel_prob` parameter of the task, and it is initialized from a configuration file during task graph instantiation. In *OR* mode `trigger_out()` function writes data sample to its output port, connected to the task with the highest probability of execution.

Since the communication FIFO channels between the nodes have finite user defined sizes, the writing to the output ports is also blocking. The task has to wait until there is space available in the FIFO channels. Once the data is written to the output ports, and if the task is a periodic task and the total number of iterations is not reached, then the task requests starting on the corresponding processing element again without reading the input ports. If the total number of iterations is reached or the task is non-periodic, it starts back from Step-2. However, in case the task is a source or autonomous task, then it ends here.

### **Task-Graph Configuration Using ISCTLM Attributes**

In our methodology, the configuration of tasks, the structure of the task graph, and the underlying architecture configuration are specified using a set of .ini extension ISCTLM initialization files. Parsing and management of these configuration files are based on Intel SystemC TLM (ISCTLM) library. ISCTLM uses the concept of attributes for the configuration of a system from .ini extension files. Our task graph also exploits this property of ISCTLM for easy and rapid configuration of the tasks and evaluation of different application structures with different mapping relations on the architecture model.

In order to be able to initialize and change parameters and configuration of a system modeled using Intel SystemC TLM (ISCTLM), the following main conditions have to be fulfilled:

- 1) The member variables of a SystemC module, whose values need to be configured from a configuration file, have to be defined as `ISCTLM Attribute<T>`, where T represents the type of these member variables.
- 2) T should be copy constructible C++ or SystemC standard data types. In order to define custom type member variables as attributes and initialize or configure them from a configuration file, some operations have to be manually defined for it.
- 3) The format of the configuration file should be according to the parser. The ISCTLM parser used for our task graph requires the correct SystemC object hierarchical name of these member variables. Furthermore, a value to a variable can be assigned using the equal(=) sign. For instance, the period of a task, named as `Task_1`, in our task graph model can be set in a configuration file as:

```
HARDWARE.TG_Constructor.Task_1.period = 20 ns
```

Where `HARDWARE` represents the top level, `TG_Constructor` is a sub-module within the `HARDWARE`, and `Task_1` is a sub-module within the `TG_Constructor` representing a task node. Figure 3.5, shows these class hierarchy (left) and the corresponding initialization file (right).

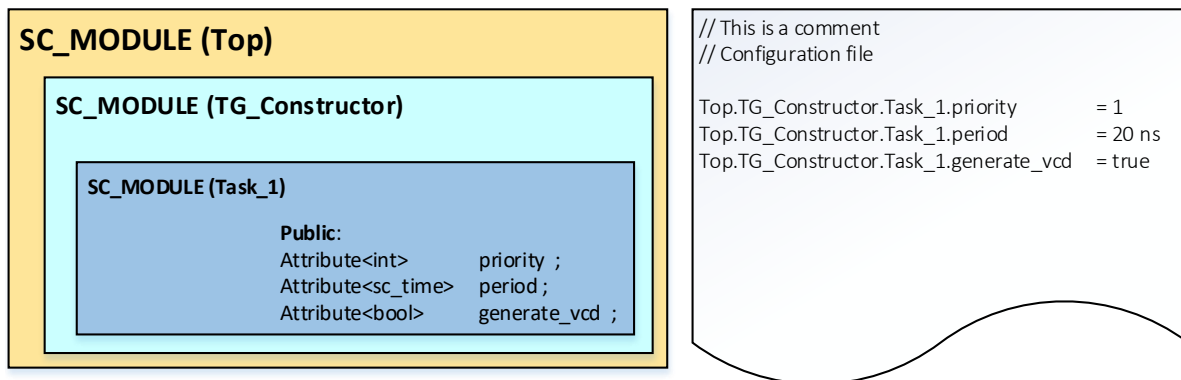


Figure 3.5: A configuration file is shown (right), consisting of three entries for task priority, period, and debug option. And the corresponding hierarchy of the class in the configuration file is shown on the left side.

### Task-Graph Creation

For task graph construction and mapping of tasks on different architecture resources, `tg_constructor` module is used. It is an `SC_MODULE` and instantiated with the task

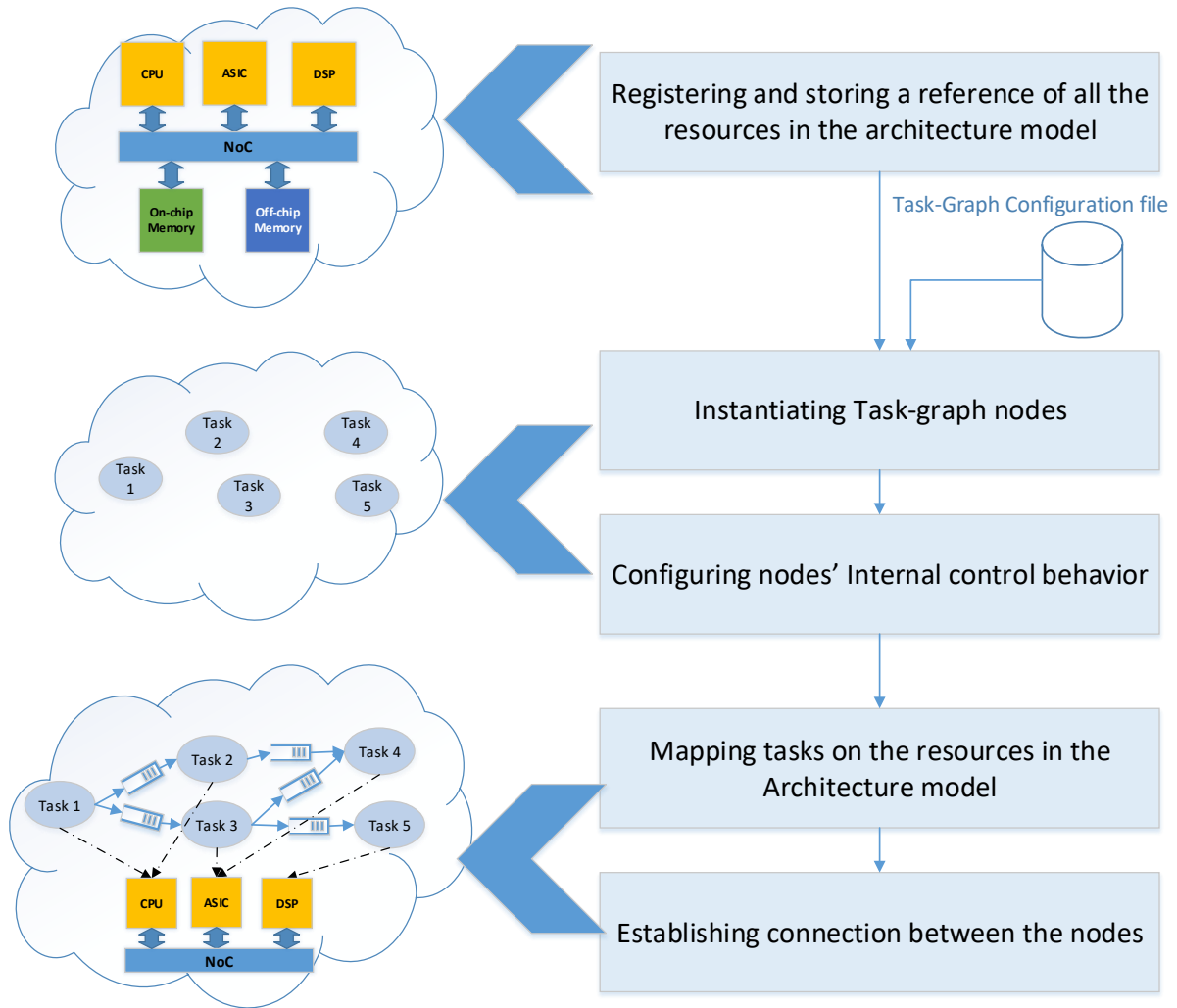


Figure 3.6: Task graph creation flow: It consists of four steps and starts with the registration of all the resources to instantiating task graph nodes to configuration nodes' internal behavior to map the task to architecture model to connection establishment.

graph configuration file reference, given as a constructor parameter. This module's functionality is to construct the task graph, map tasks on the architecture resources, collect run-time statistics about the tasks, and the underlying architecture for power estimations. It creates the task graph step by step from a configuration file according to the flow shown in Figure 3.6. The module `tg_constructor` has a set of member variables and methods as shown in Table 3.6 which are called while constructing the task graph, as follow:

**Step-1 Registering architecture resources:** Each processing resource in the architecture model, upon instantiation during the elaboration phase, calls the `hw_engine_register()` API of the `tg_constructor` module. A reference to each of these processing elements, is stored in a list (`hw_engine_list`) within the `tg_constructor`.



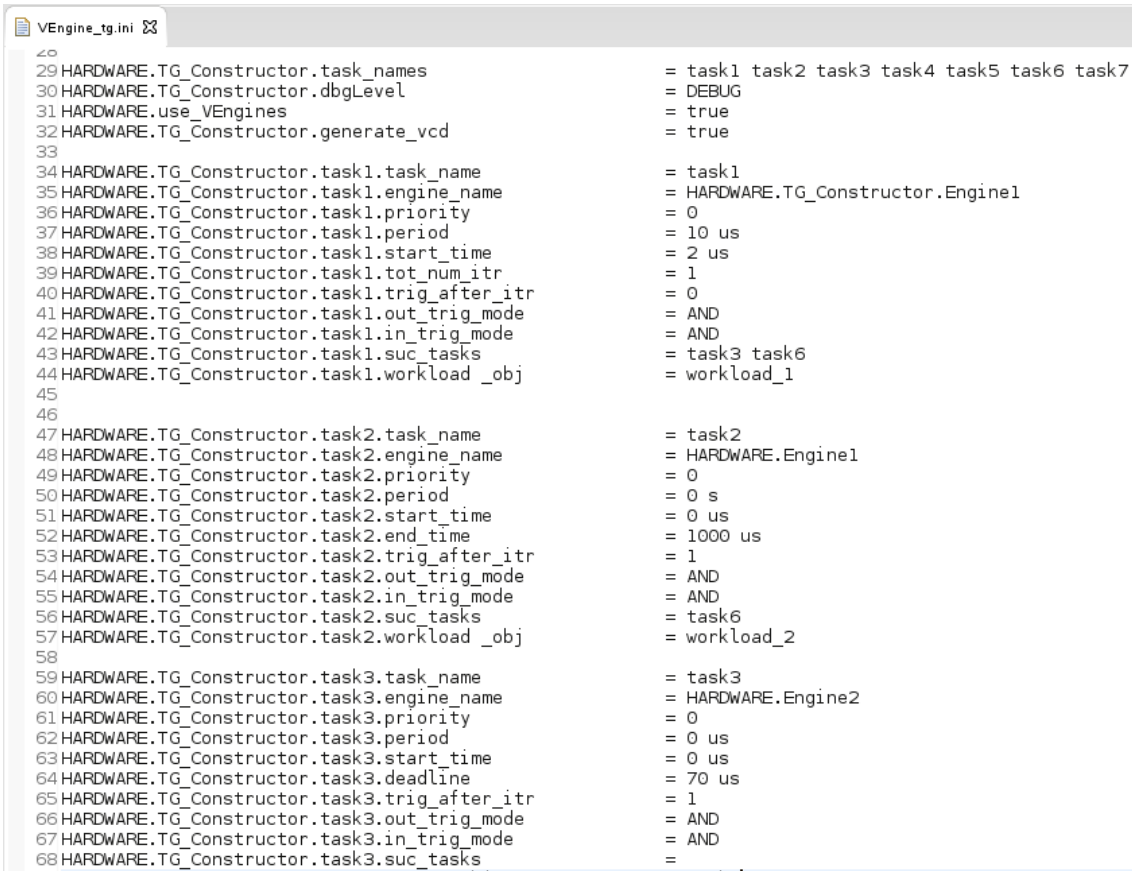
Table 3.6: Main function and variable members of `tg_constructor`

Members	Description
<code>void hw_engine_register (task_scheduler_engine_if *eng)</code>	Stores the references of processing element in <code>tg_constructor</code> .
<code>void register_all_tasks()</code>	Instantiates task nodes and invokes other function members for construction of the task graph step by step.
<code>void map_task(node* tg)</code>	Maps tasks on their corresponding processing elements or engines.
<code>std::vector&lt;std::pair &lt;int,int&gt;&gt; build_graph()</code>	Instantiates fifo channels and establishes the connection between nodes through these fifo channels.
<code>void PM_info (task_scheduler_engine_if *eng, node*)</code>	Sends necessary information to a power estimating module during simulation.
<code>void print_graph (bool write)</code>	Prints the task graph in text format in the activity report.
<code>std::vector &lt;task_scheduler_engine_if*&gt; hw_engine_lst</code>	List of instantiated architecture processing elements.
<code>std::vector &lt;tg_node*&gt; task_lst</code>	List of all instantiated nodes in the task graph.

**Step-2 Instantiating task nodes:** Before end of the elaboration phase, the function `register_all_tasks()` is called. It instantiates task graph nodes for all the tasks in the application model, as sub modules of the `tg_constructor`. The name of these task nodes are according to the names provided for `task_name` member variable of `tg_constructor` in the configuration file, as shown in line #29 of Figure 3.7. A reference to each instantiated node is stored in a list (`task_lst`). The task parameters of each node, such as mapped engine name, priority, period and name of its successor tasks are also initialized automatically after their instantiation according to the configurations provided for each task in the configuration file. For instance, `task1` is mapped on `Engine1` (line #35 of Figure 3.7), it has two successor tasks, `task3` and `task6` (line #43 of Figure 3.7), and its execution workload object(`workload_obj`) name is `workload_1` (shown in line #44 of Figure 3.7).

**Step-3 Configuring nodes:** At this step, the list of all instantiated nodes (`task_list`) is traversed and `configure_node()` function of each node is called. Which defines the internal control flow of a node according to the information provided in the configuration file. It initializes parameters and local variables, such as `iTrig_mode` and `oTrig_mode` and it also registers node's state as a SystemC `sc_logic` signal to a common trace file.

**Step-4 Mapping tasks on architecture resources:** At Step-3 each task node has only the name of its mapped engine as a string, according to the configuration file. However, it has no direct access to the corresponding engine in the architecture model. There-



```

40
29 HARDWARE.TG_Constructor.task_names           = task1 task2 task3 task4 task5 task6 task7
30 HARDWARE.TG_Constructor.dbgLevel             = DEBUG
31 HARDWARE.use_VEngines                        = true
32 HARDWARE.TG_Constructor.generate_vcd        = true
33
34 HARDWARE.TG_Constructor.task1.task_name      = task1
35 HARDWARE.TG_Constructor.task1.engine_name    = HARDWARE.TG_Constructor.Engine1
36 HARDWARE.TG_Constructor.task1.priority       = 0
37 HARDWARE.TG_Constructor.task1.period        = 10 us
38 HARDWARE.TG_Constructor.task1.start_time    = 2 us
39 HARDWARE.TG_Constructor.task1.tot_num_itr    = 1
40 HARDWARE.TG_Constructor.task1.trig_after_itr = 0
41 HARDWARE.TG_Constructor.task1.out_trig_mode  = AND
42 HARDWARE.TG_Constructor.task1.in_trig_mode  = AND
43 HARDWARE.TG_Constructor.task1.suc_tasks      = task3 task6
44 HARDWARE.TG_Constructor.task1.workload_obj   = workload_1
45
46
47 HARDWARE.TG_Constructor.task2.task_name      = task2
48 HARDWARE.TG_Constructor.task2.engine_name    = HARDWARE.Engine1
49 HARDWARE.TG_Constructor.task2.priority       = 0
50 HARDWARE.TG_Constructor.task2.period        = 0 s
51 HARDWARE.TG_Constructor.task2.start_time    = 0 us
52 HARDWARE.TG_Constructor.task2.end_time      = 1000 us
53 HARDWARE.TG_Constructor.task2.trig_after_itr = 1
54 HARDWARE.TG_Constructor.task2.out_trig_mode  = AND
55 HARDWARE.TG_Constructor.task2.in_trig_mode  = AND
56 HARDWARE.TG_Constructor.task2.suc_tasks      = task6
57 HARDWARE.TG_Constructor.task2.workload_obj   = workload_2
58
59 HARDWARE.TG_Constructor.task3.task_name      = task3
60 HARDWARE.TG_Constructor.task3.engine_name    = HARDWARE.Engine2
61 HARDWARE.TG_Constructor.task3.priority       = 0
62 HARDWARE.TG_Constructor.task3.period        = 0 us
63 HARDWARE.TG_Constructor.task3.start_time    = 0 us
64 HARDWARE.TG_Constructor.task3.deadline      = 70 us
65 HARDWARE.TG_Constructor.task3.trig_after_itr = 1
66 HARDWARE.TG_Constructor.task3.out_trig_mode  = AND
67 HARDWARE.TG_Constructor.task3.in_trig_mode  = AND
68 HARDWARE.TG_Constructor.task3.suc_tasks      =

```

Figure 3.7: A piece of configuration file depicting three tasks and their specifications using all or a subset of task graph node parameters.

fore, using the `map_tasks (node*)` function, a direct reference as pointer to the task scheduling interface of the corresponding engine is provided to the task. The function `map_tasks (node*)` searches for all the instantiated engines in the `hw_engine_list` and a reference to an engine, with its name matching to the engine name assigned for the task, is stored in the node. Using this reference, the node can directly call the API functions of the corresponding task-scheduler interface during simulation for requesting execution of its workload on the corresponding engine.

**Step-5 Establishing connection between the nodes:** The communication fifo channels between the nodes are instantiated and their connection with the input and output ports of the nodes are established. For this purpose the function `build_graph()` is used. Which traverses the list of all tasks (`task_list`) in the `tg_constructor`, for each task in the successor tasks list (`suc_tasks`) of a node. When a task name in `suc_tasks` matches the name of a task in the `task_list`, new `sc_fifo` channel with specific size, is instantiated and accordingly its connections with the corresponding nodes are established by calling the `new_inpunt()` and `new_ouput()` functions of the nodes.

All the above steps are completed during the elaboration phase of the SystemC code. The task graph structure, the mapping relation of each task, and its timing characteristics can be easily modified using the configuration file without recompilation of the task graph or the architecture model SystemC code.

In order to estimate the consumed power by the underlying architecture while running an application, power models of each architecture resources are needed. Our methodology provides the required data needed to drive the power estimation module of the underlying architecture. During a simulation, run-time statistics of each processing element is collected by the `tg_constructor` and communicated to the power module through a generic interface. This run-time statistics of a processing element in the current implementation includes information about the *states* of a processing element, whether it is *IDLE* or *RUNNING*, and if it is *RUNNING*, then at what power state which task it is executing. The characterization of the power states and power modeling of the architecture is explained in detail in Section 3.3.

## 3.2 Architecture Modeling

The architecture model comprises components from intel's system architecture validation and exploration(SAVE) framework, which has some generic and application-specific performance models for processing elements, interconnects, and memory subsystems for macro and microarchitecture exploration. These elements are implemented in SystemC using the Intel SystemC TLM (ISCTLM) libraries in our case. The processing elements in the architecture model communicate with each other through transactions via an NoC interconnect or direct point-to-point connections with different communication protocols. The operating system services are modeled as a scheduling layer on top of each processing element. We refer to this scheduling layer as *task scheduling engine interface* (`task_scheduling_engine_if`). This scheduling layer serves as an interface between the application model and the architecture model, which provides abstract real time operating services (RTOS) to the application tasks, such as resource allocation and task scheduling with different scheduling policies. Additionally, it encapsulates the execution workloads of each task mapped to the corresponding processing element. Like the task graph, the run-time configuration of these elements and the workload objects are defined by a configuration file.

### 3.2.1 Task Scheduling and Resource Allocation

The task scheduling engine interface is modeled as a SystemC Module, which forms a base for all the processing elements in the architecture models including the virtual-engine. As shown in Figure 3.8, it has some pure virtual function members, such as `start_task()`, `preempt_task()`, and `resume_task()`, which have to be implemented inside the corresponding processing elements. These functions represent the APIs for the task-scheduling layer, to start, preempt or resume task executions during a simulation on a processing element. Additionally, the scheduling layer provides an API, `req_start_task()`, to the task nodes in the task graph model, which can be accessed by each task for requesting execution on a

processing element.

The following are the primary services provided by this task-scheduling engine interface:

- *Scheduling*: Many tasks that might be running in parallel can be mapped on the same processing element. The process of deciding the order of execution of these parallel tasks on the corresponding processing element is called Scheduling. In this scheduling interface, the tasks are scheduled according to their priorities. Nevertheless, the implementation of this scheduler is generic enough to allow integration of other scheduling policies as well.
- *Resource allocation*: Depending on the type of resources, a task can be granted access to a resource until it is completed (run-to completion (RTC)), or it can be preempted by another higher priority task and rescheduled for resuming later. Therefore, the resource allocation policy can be RTC or preemption based.
- *Context switching*: In the case of preemptive resource allocation, the execution cost of switching between tasks is simulated by the `context_switch` block within the task-scheduler interface. This cost accounts for context storing or context loading of a preempted task.
- *Task execution workload modeling and instantiation*: As discussed in Section 3.1.1, workload objects are basically a different set of values for the run-time parameters of a processing element. Executing different tasks on a processing element means running the same processing element with different values assigned to its run-time parameters. Therefore, before starting the execution of a task on a processing element, these run-time parameters of the corresponding processing element need to be initialized with values according to the task's execution workload object. It has to be mentioned that not all the run-time parameters of a processing element are task-dependent. Therefore, the workload objects (`workload_obj`) only store the task-dependent parameters of a processing element with their values. These execution workload objects are instantiated inside the processing elements as SystemC sub-modules. The member variables of these workload objects, which are the corresponding processing element's run-time parameters, are defined as `ISCTLM Attribute<T>s`. Therefore, they can be initialized with specific values from a configuration file. The scheduling layer provides an API to the task graph constructor (`tg_constructor`) for instantiating these workload objects during the task graph construction. The number of these workload objects depends on the number of tasks mapped on to the same processing element but with different execution loads. When a task requests starting on a processing element, the corresponding scheduling interface binds it to its workload object (`workload_obj`) and then schedule it for execution. When the scheduler dispatches a scheduled task, the processing element's run-time parameters are initialized according to the corresponding workload object.

During a simulation, when a task requests access to a resource, it goes through the following steps:

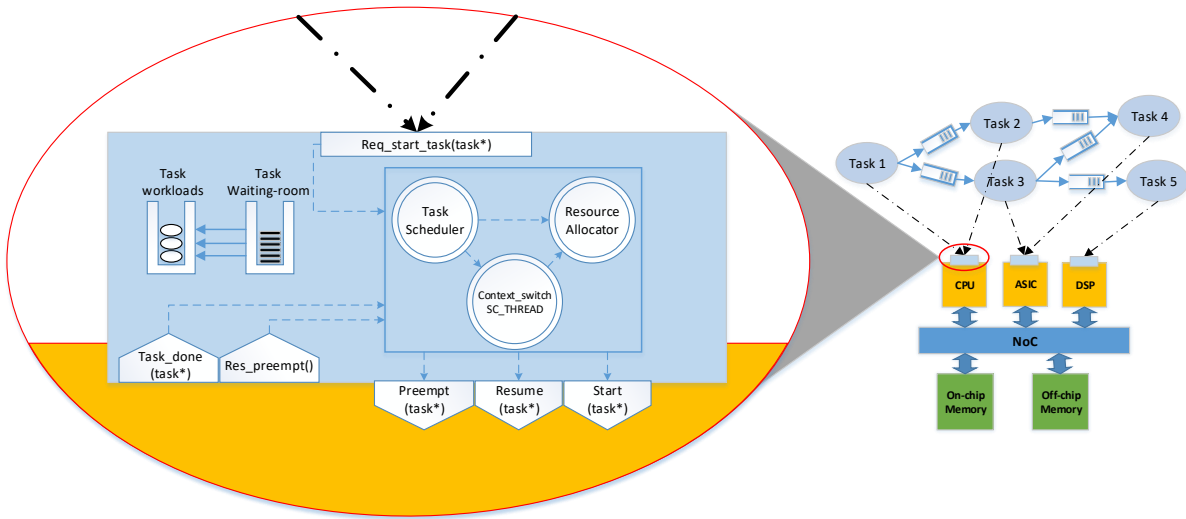


Figure 3.8: An enhanced (internal) view of the Task Scheduler Interface. It acts as a Real-time operating system as it controls the functionality and manages the data-flow between the task graph nodes and processing elements.

Step-1 When a task in the task graph is triggered, it requests executing on its mapped processing element by calling the `req_start_task(task*)` API of the corresponding task scheduling interface of a processing element. `req_start_task(task*)` binds the reference of the corresponding `workload_obj` to the task and asks the scheduler block for scheduling the execution of the task. The scheduler places the task in a waiting-room according to its priority.

Step-2 Upon receiving notification from the scheduler block when a task is dispatched, the context switch block asks for resource allocation and, if necessary, simulates the context switching cost of the task by consuming specific simulation time.

Step-3 The resource allocation block checks the state of the current processing elements. If it is IDLE, then the resource is granted, and the execution of the task is requested by calling the `start_task(task*)` function, which is implemented inside the processing element. If the resource is occupied by another task having a lower priority, then a preemption of this other task is requested by calling the `req_preempt()` function.

Step-4 When a processing element completes the execution of a task, it calls the scheduling interface's function `task_done()`. The scheduler dispatches the next waiting task for execution. In case preemption of a task was requested in Step-3 and the task is preemptible, then instead of calling `task_done()`, the processing element calls the `resp_preempt()` API of the scheduling interface. In this case, the preempted task is rescheduled, and the scheduler dispatches another task with a higher priority.

### 3.2.2 Virtual-Engine

Virtual-engine processing element is an `SC_MODULE`, which is the SAVE framework's most basic performance model. It has the same task scheduling interface as the other processing elements of the SAVE framework, shown in Figure 3.8, for scheduling of its mapped tasks. It is mainly used to analyze and develop the application structure and configuration of the application tasks in the task graph model. It can also be used in combination with other SAVE architecture resources for executing tasks, whose execution does not require the generation of any traffic or memory access or where only the execution time of the tasks and the processing resource contentions are concerned. Therefore, it has no SystemC TLM sockets or ports to interact with other resources in the architecture model. It implements the basic pure virtual functions of the task scheduling interface, such as `start_task(task*)`, `resume_task(task*)`, and `Preempt_task(task*)`. Additionally, it has its internal behavior, which is abstractly shown in Figure 3.9.

The execution of a task is simulated using the SystemC `wait(exe_time)` statement. The waiting time, `exe_time`, depends on the task's execution time. A running task can be preempted by adding a SystemC event (`sc_event`) as a second parameter to the wait statement (`wait(exe_time, preempt_eve)`). Whenever, a preemption is requested the running task's remaining `exe_time` is stored in its workload and engine waits for being triggered again by its scheduling layer. When a preempted task is resumed back, it only waits for the corresponding remaining time of its total execution.

## 3.3 Power Modeling

Modeling in the context of embedded systems is a technique to accurately define vital processing elements, i.e., processor, hardware accelerator, memories, and buses in a theoretical manner to better understand their functionality. Thus, power modeling estimates the power consumption of the processing elements and develop power optimization policies to optimize the system.

To analyze power models, we need to answer three fundamental questions; *How accurate it is? How fast is it? How much effort is required?* These characteristics are conflicting in nature, and a trade-off is required to achieve an optimal solution.

It is desirable to model the embedded system as accurate as possible. For accurate modeling, design details are pre-requisite. Design or implementation details evolve over the time-line of the product development cycle. Hence, there is a trade-off between how early we want to develop power models and how accurately we want to develop them. Higher the abstraction level, lower the accuracy of the power model.

Ideally, power model execution should be fast so that the system's power consumption is available in no time. Fast execution allows exploring different aspects of the system, i.e., the impact of changing frequency and voltage. The cost of fast execution is less control over the system's observation points, which results in inaccurate power estimation.

Power modeling efforts depends on the available implementation details and the abstraction level. Lower power modeling efforts are attractive as design space exploration (i.e.,

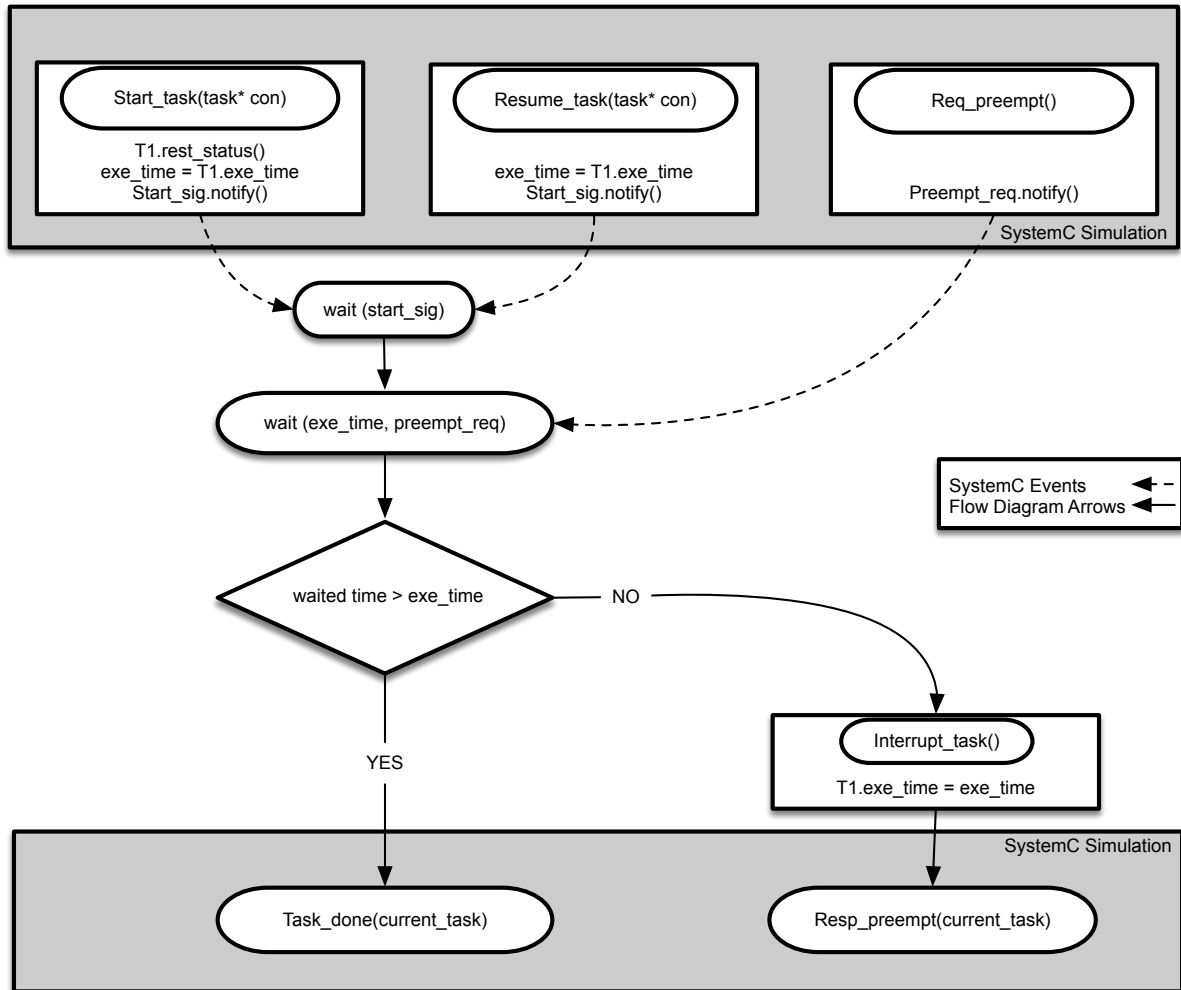


Figure 3.9: Virtual Engine processing element internal flow: In a Virtual Engine, the execution flow starts with a wait state. The start signal notifies about the task to be simulated. It could be a new task or an already running task that resumes. The wait statement in SystemC simulates the execution time of the current task in a Virtual Engine. The wait state of a task graph node is preemptable by a SystemC event, given that the execution time is greater than the current waited time at the point of the SystemC event preempting the task. As soon as the waited time is greater than the execution time, it ends.

implementations of application on different design configurations) can be achieved in a faster manner. Higher the power modeling efforts, the more significant the time required to explore different design options.

The power consumption of SoCs is calculated by static and dynamic power consumption. The static power is contributed mainly because of leakage, irrespective of the computation being performed by the SoC. In contrast, the switching activities contribute to the dynamic

power discussed in 2.1.4. Heterogeneous SoCs are becoming rapidly complex with every passing day, and conventional methods used by system engineers to generate crude powers number at the early stage of the design process, by maintaining excel sheets and doing an extrapolation of in-hand data from previous generation platforms, is neither suitable nor error-prone anymore. Power models need to develop as soon as the basic system requirements are available, as the challenging task of hardware and software partitioning be carried out before the register transfer level (RTL) is frozen.

Power modeling approaches to develop high-level power models have been discussed in detail in Section 2.1.7. We use the power state machine (PSM) that complements POEM's multi-abstraction level, modular, and holistic approach. It resembles the approaches described in [86, 87, 88] as they are also architecture-level estimation approaches and use inbuilt processor performance counters to estimate the energy.

Some of the described frameworks in Section 2.2.2 do not specify how to create power models. Others are specific to only one type of component or require electronics system level models created according to a specific modeling approach. In contrast, this work presents a generic methodology for creating electronics system level power models for components of any type. Further, it applies to white box electronics system level models created with any modeling style and black box electronics system level models.

The power modeling framework used is an electronic design automation (EDA) tool called Intel® Docea™ [89] and previously known as Aceptlorer. The motivation behind using Docea™ is its ability to target architecture-level power modeling and thermal analysis along with flexible options to connect with an external event-based simulator. The interface options include application programmable interface (API)s or a value change dump (VCD) file.

### 3.3.1 Docea™

Stand-alone power models simulation for handwritten scenarios can give pessimistic power numbers for a scenario, but the use of SoC for real time operating services is highly dynamic, and handwritten scenarios can not cover all the system level use cases. Another practical challenge faced during power modeling is managing, structuring, and collaboration of huge power modeling data. It can range from component characterization, leakage dependencies law, abstraction of power domains activity factor, till writing system level uses cases.

The Intel® Docea™ power and thermal modeling and simulation solutions provide an analytics and a simulator, called Intel® Docea™ power analytics (IDPA) and Intel® Docea™ power simulator (IDPS) respectively can handle above challenges. "It is a web-based, collaborative power modeling and simulation framework that improves power roll-up productivity and early power architecture exploration" [89].

#### IDPA

Power models for the components are developed in IDPA by populating them with power design data. PSM-based approach for design space exploration and early power analysis in [90] are used to model components in IDPA. PSM allows components to have a hierarchical



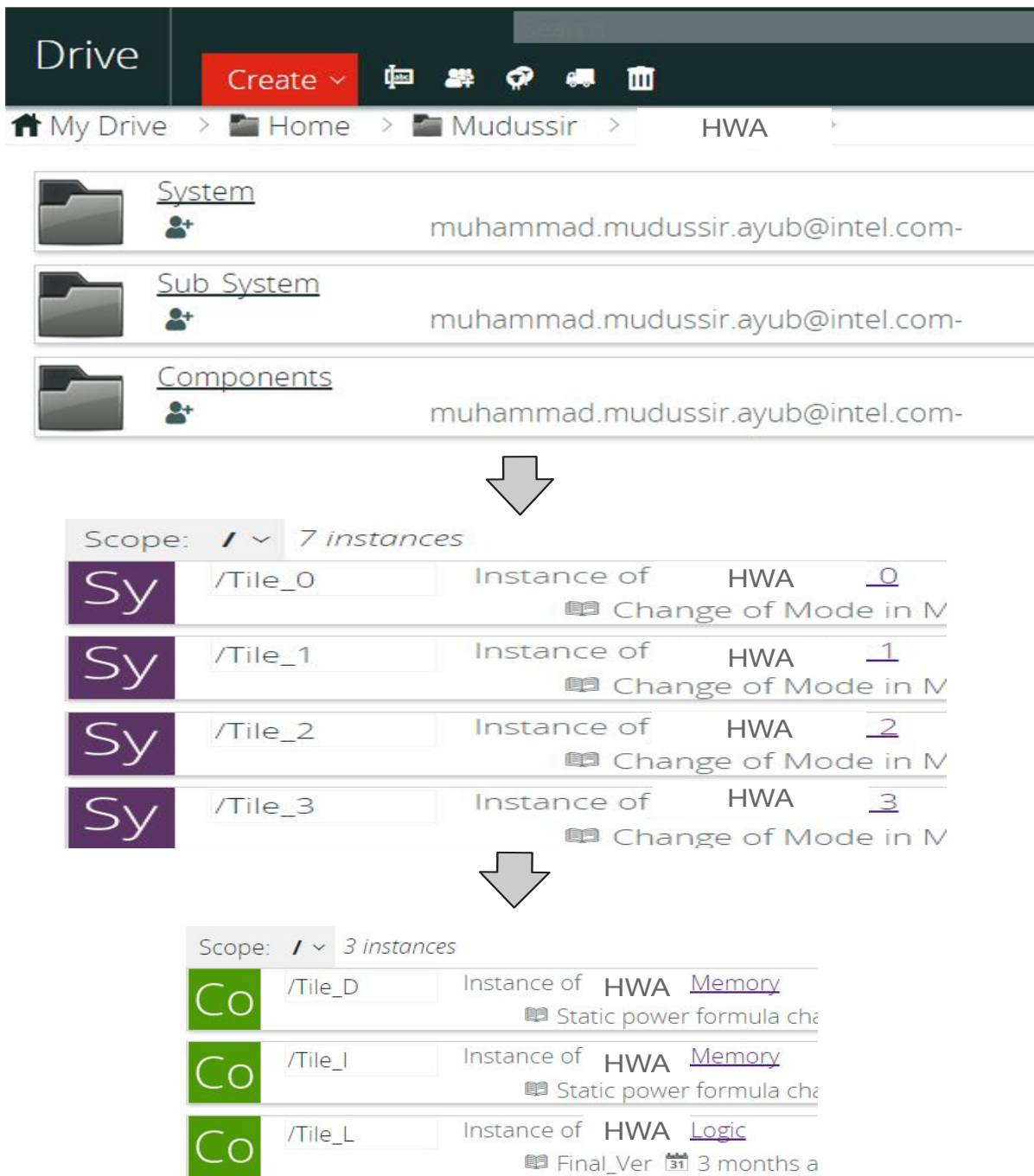


Figure 3.10: Docea flow: At the top of the hierarchy sits System, Subsystem, or Component. A System contains the Subsystems or Components to give the flexibility and ease of modeling components and reusability in different projects.

structure, which makes them suitable for system level integration and to back-annotate the low-level power information by creating sub-components as shown in Figure 3.10.

IDPA provides flexibility to model a system, subsystem, or component with the input, user-defined parameters, modes, and basic formulas. The input parameters are the voltage and clock frequency. User-defined parameters include the process, temperature, and any other parameter helpful to model a component, i.e., capacitance value, activity ratio, frequency divider, and memory dimensions.

Modes in IDPA are used to characterize the components like logic and memory. Modes can vary from components to components, but the most commonly used modes are Active, Clocked, Idle, Retention, and Off.

IDPA calculates the dynamic and static power at the component level using relevant parameters and the currently selected mode. IDPA also provides a modeling component called a database that can be populated for third-party IP and increase power models' re-usability across different projects and systems.

In IDPA, power modeling for any component is done hierarchically. It starts with the system level, then follows the component level, and behind each component, a formula is attached to calculate the consumed power depending on the mode. Power data needed to build power models in IDPA can come from the component data-sheets, previous-generation platform, RTL simulations, or can be inferred by the component's design and functional scaling. The power number reported in IDPA are average power numbers.

## **IDPS**

IDPS, a software solution provided along IDPA by Intel® Docea™ to couple power model with performance models, and to run realistic power simulations for system uses cases.

IDPS simulates the system level use, scenario, or application running on multiprocessor system on chip (MPSoC) for dynamic power Analysis. There are multiple ways to connect the performance model with IDPS. As the definition of the performance model has already been explained in Section 2.3, it simulates the computation and communication workloads of the use case or system scenario with the main focus on data volume rather than data content. The stimuli generated as a result of the performance model are re-run or re-stimulated in IDPS.

IDPS is an external power solver that is driven by events. Different types of performance models can produce these events at different abstraction levels. The interface of IDPS is flexible and generic in the sense that it can be connected with any event-driven performance model that captures multiprocessors system on chips functionality and the state-residencies of the critical components in the system.

The power model is defined in IDPA and can be exported in Python (as a package, an .egg folder) for dynamic simulations. By default, the static parameters at the power model interface can be stimulated in a dynamic simulation.

If power model internal signals are to be modified or observed during the simulation, the configuration file must be used. The configuration file is a JavaScript Object Notation (JSON) file with the following fields: " input " contains the definition of new inputs exposed at the interface of power model.

```
1 "<input name>" : {" params " : [ <list of path to power model parameters>]} "
   outputs "
```

And it also contains the definition of new outputs exposed at the interface of the power model.

```
1 "<output name>" : {" params " : [ <list of path to power model parameters>]} "
```

```
1 "<input name>" :{
2   " params " : [<list of path to power model parameters>],
3   " type " : <"str", "init", "float", "bool">,
4   " default " : <default input value at init>,
5   " values " : [<list of expressions to convert an input value to a power model
6   parameter value>]
```

Listing 3.1: Definition of input with value conversion, It is possible to connect an input to power model parameters and convert input values for each power model parameter. In the expression, the input value is identified by the tag input.

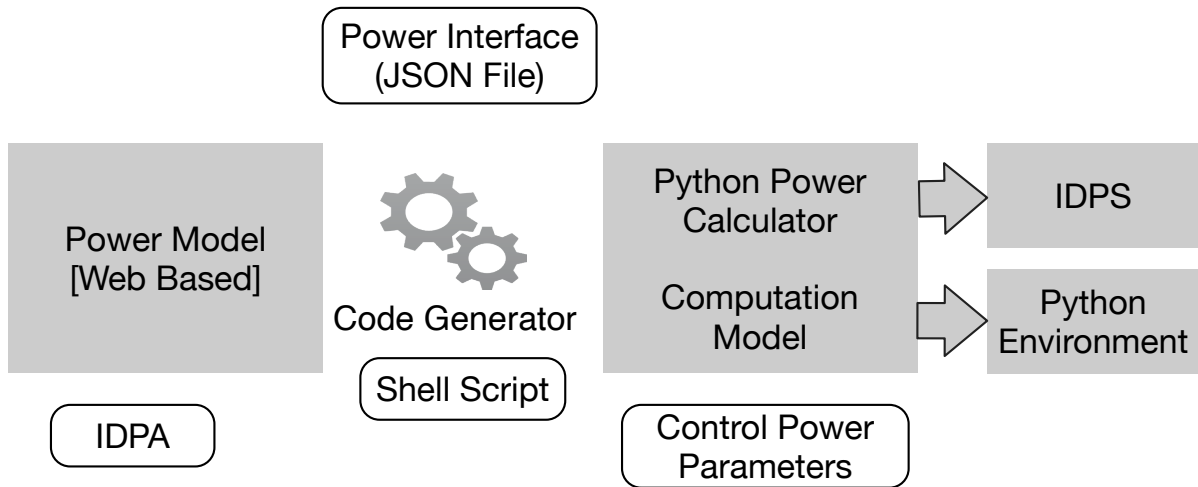


Figure 3.11: Generation of computable power model from a web-based Docea™ Power Analytics: An offline script produces a computational model using power interface information in the form of a JSON file. It is also seen as a Python power calculator. It gives in-feed to Docea™ Power Simulator and Python environment configurable by control power input parameters, i.e., voltage, frequency, modes, activity ratio. [89]

The overall flow of the Docea™ is shown in Figure 3.11. An offline script converts the power model developed in IDPA into an executable project (computation model) of the user's desired language, i.e., Python or C++. The interface file establishes the performance model's

connection with IDPS by exposing and connecting the parameters of the power model in IDPA with the performance model. IDPS has two options: offline in the form of value change dump or integrated with the application programming interface (API). In this dissertation, the offline approach is more suitable to keep in line with the modular setup of POEM.

IDPS generates an output in a value change dump file that users can visualize to see the power trace over time or even opt for post-processing.

## 4 Implementation of the Methodology at Architectural Abstraction Level

This chapter will focus on implementing the power and performance optimization and exploration methodology (POEM) at architectural abstraction level (AAL). The reason behind targeting this abstraction level is missing implementation details of the application, performance, and power model. The design space exploration is vast and challenging to secure specifications, optimize power behavior, and accelerate validation and verification of design projects at architectural abstraction level.

We have introduced application, architecture, and power modeling in Chapter 3. The interface and mapping between the performance model (driven by the application model) and power model are described using hardware accelerator. This chapter gives an overview of the methodology first, followed by the implementation details of the application, performance, and power models; at the end, the interface and mapping have been explained.

### 4.1 Methodology: Overview

Our methodology POEM at architectural abstraction level aims to provide reliable system level power and performance estimates of a target multi-core embedded system architecture for an abstract use case or application. It does not require the detailed implementation of the application or the target architecture at the architectural abstraction level. Therefore it enables the system architects to explore the power and performance of different design choices at the system level in the early phases of the development process.

In this methodology, following the Y-Chart [91] approach, application and architecture models are developed independently. The application model provides abstract execution workloads of application tasks to the underlying architecture while capturing the application's correct control and data flow.

In POEM, an application or a system level use case is modeled as a task graph using the Intel SystemC TLM (ISCTLM) library (a productivity layer) on top of Accellera [83] standard SystemC.

Likewise, all the processing elements of the performance model, i.e., processors, memories, buses, are modeled at transaction level with approximately-timed (AT) coding style using the same ISCTLM library. ISCTLM is compliant to SystemC standards and includes generic bus sockets, component base classes, power trace infrastructure, register modeling tools, and many other convenience modeling tools for efficient AT modeling and simulation. The simulation environment of POEM (shown in Figure. 4.1) comprises of the performance model, which is driven by the application model.

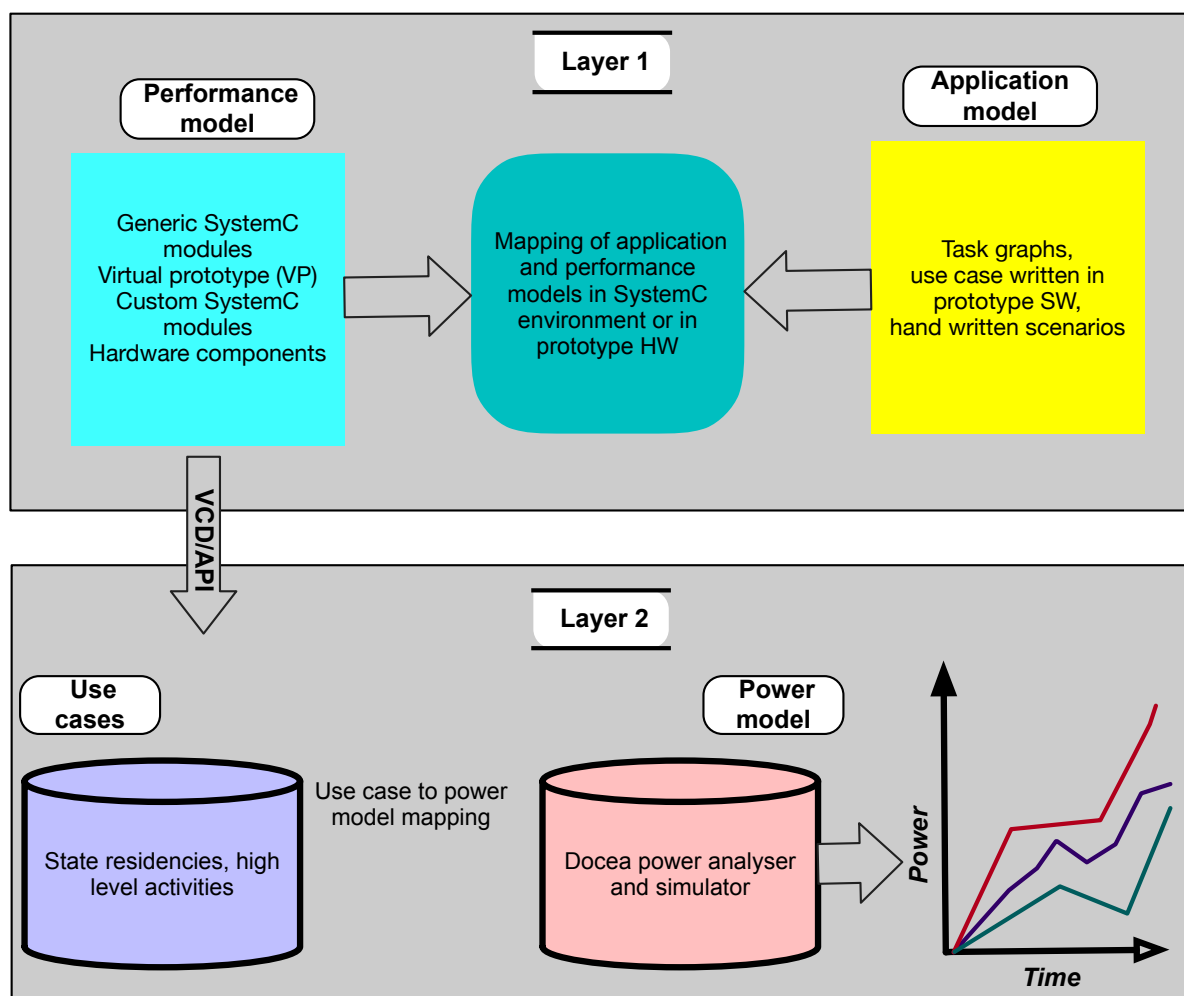


Figure 4.1: POEM: Layer-1) application modeled as a task graph mapped to processing elements of the performance model. Layer-2) SystemC simulation of Layer-1 generates performance and power stimuli, mapped to the power database, and re-simulated for power vs. time for different configurations.

The application model tells the performance model to simulate each application task's computation and communication workload and acknowledge it as it finishes. The behavior of the SystemC simulation can be controlled using attributes based configuration mechanism, as the application and performance models are configured dynamically using these attributes.

At the end of the simulation, stimuli files are generated, which contain the functional states, and timing information, together known as the state residency of a processing element. The power models are developed in Docea™ (not SystemC), a mapping step and an interface are required to drive them using the stimuli generated from SystemC simulation for dynamic power estimation. The power trace over time for different design choices along performance analysis gives system designers a complete picture to co-optimize the embedded system for

power and performance.

#### 4.1.1 Implementation of Application Model

The application model is developed as task graphs, and during the simulation, it provides the computation and communication workload of use case's tasks mapped to the performance model. The task graphs capture the application at a high level of abstraction, such as a group of interacting processes or tasks. The task graphs nodes represent abstract application tasks, and the edges represent the dependencies and the communication channels between the tasks. Each task, represented via a node, can have input and output ports connected with other nodes through a SystemC FIFO channel. The nodes communicate via these channels by sending tokens to each other.

A node is triggered when it receives a token from its predecessor nodes, and then it requests access to its mapped processing element to execute. By executing an application task on a processing element in the performance model at architectural abstraction level, we mean simulating the communication and computation workload (data traffic generation and simulation time consumption) of the task by the corresponding performance model. The access response from the resource (processing element) side depends on the corresponding resource's load and the priority of the task itself. If the resource is free, it will start simulating the corresponding task's workload; otherwise, it will put the task in a pending state and schedule its workload simulation according to the number of its other pending tasks and their priorities.

Suppose the requesting task has a higher priority than the already running task, and the running task is preemptible. In that case, the processing element can preempt the execution of its current task and starts the new task. When the execution of a task is completed on a processing element, it notifies the corresponding node. Upon receiving notification of completion, the node triggers its successor's nodes by pushing tokens to its output channels via its output ports.

Each node has a set of parameters for specifying its behavior, its workload on a processing element, and its interaction with other nodes.

Some of the parameters needed in the described interactions are listed below:

- 1) Start time: If a node has no predecessors, then a specific value can be configured as its starting time.
- 2) Period: A task can be configured to run periodically till the end of the simulation or for a specific number of iterations.
- 3) Triggering mode: This parameter is used to decide whether a node should start after it receives a token on one or all of its input ports and in which order these tokens should be consumed. Similarly, the triggering mode of its successor nodes can also be configured using a similar parameter.
- 4) Priority: Each task can be assigned a priority for accessing performance model resources.

- 5) Iterations/trig: It is used to specify how many iterations a node's task needs to complete before triggering its successor nodes.

### Application Stand-Alone Simulation Using Virtual Engines

An application's task graphs model can be simulated stand-alone to analyze the application itself and find initial design configurations for the target architecture. In the stand-alone simulation, each task is mapped on a set of virtual processing elements referred to as *Virtual Engines* in this work. The execution workload of a task, when mapped on a *Virtual Engine*, is the estimated non-interrupted execution time (`exe_time`) of the task on the target processing element. This execution time can be estimated according to the information in the corresponding target processing element's datasheet. If the engine represents a CPU, then it can be estimated by the approximate number of instructions needed for executing the corresponding task on the corresponding CPU.

At the end of the simulation, an activity report is generated, giving detailed information about each task's run-time statistics. This information contains the minimum, maximum, and average execution time (considering the execution time stretching because of resource contentions), completion time, deadline misses, back-pressure by the successor tasks. Stand-alone simulation results can also answer questions like how many programmable or dedicated hardware blocks are needed and an initial partitioning of the application. Partitioning means implementing some tasks as software instructions running on a programmable resource and some tasks as hardware descriptions for dedicated hardware resources.

#### 4.1.2 Implementation of Performance Model

POEM uses generic performance models for key architecture components such as CPU, memories, interconnects, and some other functional blocks, fundamentally the same as discussed in [92]. The performance model of CPU uses a stochastic modeling approach for data generation and mimicking the behavior of a software task, i.e., generating a traffic pattern for instruction and data cache miss and consuming simulation time for representing the execution of a specific number of instructions. This approach is suitable for architecture design space exploration before the software is available, and the platform depicted in Fig. 4.2 is not meant to run real modem software.

The performance model simulates computation and communication workloads of application tasks using non-blocking transactions based on SystemC TLM communication. The main focus and relevance of performance models are on the data volume rather than the actual data content.

For instance, let us consider an hardware accelerator performance model and a network processor application task called packet-classification mapped onto it. The performance model of this hardware accelerator simulates the execution time of this packet classification task and also sends transactions of different data-lengths on its interfaces for representing the resulting traffic (reads/writes) of the corresponding packet data to/from the memory models, respectively.



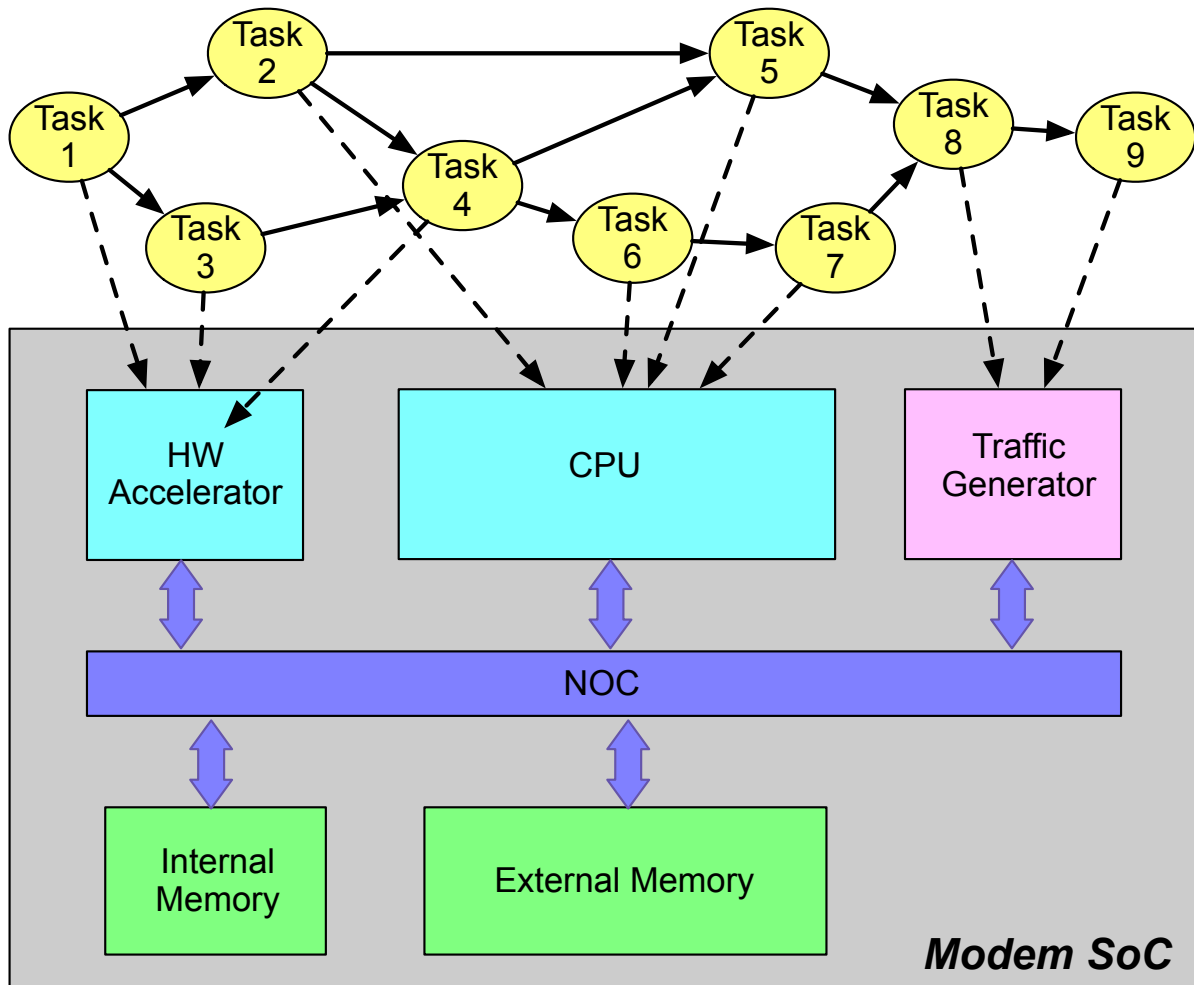


Figure 4.2: Abstract representation of main functional blocks of a cellular modem (below) and of a task graph mapped to the blocks (above).

Each processing element (Engine) in the performance model has a task scheduling interface which schedules the execution of parallel tasks on the same processing element. It also serves as an interface layer between a processing element and the task graphs nodes in the application model. For performance analysis, different arbitration policies are implemented as the interface is event-driven. Similarly, performance model allows annotation of resource access, processing latencies along with modeling of back pressure that is crucial for performance analysis.

### Mapping Application Tasks on Architecture Model Resources

A task graph is mapped on an architecture model to analyze its behavior and identify design bottlenecks of the architecture model for different mapping configurations. In the mapped simulation, each task in the task graph is mapped on one of the processing resources in

the architecture model which are referred to as *Engines* as well in this dissertation, through the configuration files. During the simulation, the tasks request executing on their mapped architecture resources (Engines). The resources while executing the tasks (simulating only the abstract execution workloads of the tasks), generate some traffic and upon completion notifies the task node in the application model.

As shown in Figure 4.2, many tasks can be mapped on one engine. Their execution is scheduled by the corresponding engine's task scheduling layer. The total execution and completion time of a task depends on the load of the engine, load, and latency of the interconnect and memory subsystems.

### 4.1.3 Implementation of Power Model

There are two fundamental questions associated with power modeling, which we have to answer before expecting accurate power estimation and optimization at electronics system level (ESL). How to develop the power model in the first place? Moreover, how to simulate the power model from electronics system level simulation?

Analytical (power state machines (PSM)) and regression-based (linear regression and beyond) modeling techniques are available to answer the first question as discussed in section 2.1.7. We have opted for the analytical approach, as it allows separation between the simulation of the functionality and the power modeling, thus, in-line with the separation of concerns principle.

Multiprocessors system on chip consists of digital and analog blocks; analog blocks are challenging to model because of their non-linear nature and their dependence on functionality. One way to represent an analog block is to consider it as a black box and specify the output power consumption for specific input by creating a database, i.e., look-up tables.

Digital blocks, on the other hand, can be divided into logic and memory parts, and their characterization is possible with different power states. In this work, we assume the following power states for logic: Active (executing a task), Clocked (waiting for the resources to be available to execute a task), Idle (a low power consumption state, no task is executing neither scheduled) and OFF. Similarly, memory has Active (data is being either written or read), Clocked (the component is in use, but memory is not accessed), Retention (component not in use, a low power state to retain the content), and OFF states.

The power consumption of the multiprocessors system on chips consists of dynamic power and static power. The leakage current because of the imperfections of the silicon is the main contributor to static power. If the area and silicon technology node (i.e., standard cell types, leakage reference data) of the component are known, we can easily calculate the static power with the help of area metrics. Leakage base value tables for different variants of standard cells (fast, slow, and nominal) in a unit area are provided by the silicon technology node.

Dynamic power consumption is because of the charging and discharging of the capacitances in a circuit. For Active and Clocked state it can be modeled as

$$P_{dynamic} = \alpha \cdot C \cdot V_{dd}^2 \cdot f, \quad (4.1)$$

where  $P_{dynamic}$  is the dynamic or active power,  $\alpha$  is activity ratio,  $C$  is the capacitance,  $V_{dd}$

is the supply voltage and  $f$  is the clock frequency of the component. The product of  $\alpha$  and  $C$  is called switched capacitance ( $C_{dyn}$ ) and it varies for Active and Clocked state. The  $C_{dyn}$  for each power state can come from component data sheets, previous platform generations, register transfer level simulations, or can be inferred by the design and functional scaling of the component.

Active, Clocked, Idle/Retention is just one basic representation of power states for a component. In principle, what is needed is a  $C_{dyn}$  per power state (the number of power states depends on the component), or in certain cases an equation of  $C_{dyn} = F(\text{system parameters})$  where a fixed  $C_{dyn}$  value cannot be given per power state. For example  $C_{dyn}$  in Active state might depend on certain data pattern, or data throughput.

There are different methods to integrate PSM to electronics system level simulations. An XML file containing the power database is attached with SystemC simulation in [40], PSM state transition can come directly from functional simulation calls, or power state can be derived by observing the activity of the processing elements. We use the later as it is a more sophisticated approach, and it allows us to build power models at different abstraction levels; hence, even more independently from performance models.

We annotate PSM with switched capacitance instead of static value as in [93] so that same PSM can be used for different frequencies and voltages. Thus, power optimization techniques like dynamic voltage and frequency scaling (DVFS) can be implemented without any modifications. We use a well known electronics system level industrial framework called Docea to create power database and simulate it. Docea is an attractive choice for our work because this framework provides a simulator called Intel® Docea™ power simulator (IDPS) and works on the principle of PSM. Details of Intel® Docea™ have already been discussed in section 3.3.1.

#### 4.1.4 Interface and Mapping

The performance model runs a system level use case and produces a trace file containing the information of resource utilization and execution latencies (functional states along with timestamps). This information can be mapped to power models with the help of an interface/connection between performance and power models.

The correlation of power data with performance models is critical to get the right balance between modeling efforts and estimation accuracy. Power models can be coarse grain and less accurate or fine grain and more accurate. Mapping and back-annotating the design data to a performance model can increase the accuracy of the estimated power. In the POEM, the mapping of the power data to performance models is done manually and by a close collaboration during the development phase of the performance model and power model.

There are two primary techniques to build an interface between performance (SystemC simulation) and power model, i.e., using an API or value change dump file. Value change dump file, an offline simulation technique, has its advantages and disadvantages, as it allows running power simulations for refined power models independently of the task graph and performance model changes, but it slows down the functional simulation. API based simulation is faster and allows a feedback loop for the system optimization. For this work,

we use a value change dump file although API can also be used similarly.

The number of components in performance and power models are not equal most of the time as they are created independently at different abstraction levels and granularities. The performance model could be fine-grained, whereas the power model can be coarse-grained and vice versa. Hence, the connection of such models is a non-trivial task and requires an explicit mapping definition. The modularity and hierarchical nature of IDPA allow the structuring of power models to be developed and mapped to performance models later in time.

Suppose the system has  $M$  components in the performance model, and each component  $c_i \in \{1, 2, \dots, M\}$  can take  $K_{c_i}$  discrete functional states,  $S_{c_i} \in \{1, 2, \dots, K_{c_i}\}$  at a given time of point. Given  $N$  number of time stamps  $t_j$  in total simulation time  $t$ , the functional states of the components in the performance model can be defined as  $M \times N$  matrix

$$\mathbf{W} = \begin{bmatrix} S_{c_1, t_1} & S_{c_1, t_{j+1}} & \cdots & S_{c_1, t_N} \\ S_{c_{i+1}, t_1} & S_{c_{i+1}, t_{j+1}} & \cdots & S_{c_{i+1}, t_N} \\ \vdots & \vdots & \ddots & \vdots \\ S_{c_M, t_1} & S_{c_M, t_{j+1}} & \cdots & S_{c_M, t_N} \end{bmatrix} \quad (4.2)$$

Similarly, the power states of the components in the power model can be shown in a  $Q \times N$  matrix  $\mathbf{Y}$

$$\mathbf{Y} = \begin{bmatrix} P_{r_1, t_1} & P_{r_1, t_{j+1}} & \cdots & P_{r_1, t_N} \\ P_{r_{i+1}, t_1} & P_{r_{i+1}, t_{j+1}} & \cdots & P_{r_{i+1}, t_N} \\ \vdots & \vdots & \ddots & \vdots \\ P_{r_Q, t_1} & P_{r_Q, t_{j+1}} & \cdots & P_{r_Q, t_N} \end{bmatrix} \quad (4.3)$$

Assuming that  $Q$  is the number of components in the power model, that each component  $r_i \in \{1, 2, \dots, Q\}$  can have  $L_{r_i}$  discrete power states, and  $P_{r_i} \in \{1, 2, \dots, L_{r_i}\}$  at a given time stamp.

The operation of extracting power states from corresponding functional states is performed by the mapping function  $F(\cdot)$ , associating a set of power modes  $P_{r_i, t}$  from the history of functional states  $S_{c_i, t'}, t' \in \{1 \dots t_N\}$  for each time stamp.

$$\mathbf{Y} = F(\mathbf{W}) \quad (4.4)$$

The mapping function  $F(\cdot)$  is written manually once for every new component, and the next section explains how it is obtained for an hardware accelerator.

### Hardware Accelerator Mapping Example

Hardware accelerators in the SoCs help the CPU to manage the traffic load efficiently; the building blocks of their architecture are called tiles (also known as cores) of different sizes in this dissertation. These tiles are made of sub-components like logic and memory, and memory is further subdivided into data and instruction cache.

Performance model with the help of task graphs maps the system level use case on hardware accelerator, and it can take different functional states e.g. Burst\_Process, Clocked\_ON,

Table 4.1: Distinct functional states of a tile and corresponding power states

Functional states $S_{c_1}, (c_1 = \text{Tile}_0)$	Power states ( $\text{Tile}_0 = P_{r_1} + P_{r_2} + P_{r_3}$ )		
	$P_{r_1} = \text{Logic}$	$P_{r_2} = \text{Data Mem}$	$P_{r_3} = \text{Inst Mem}$
Power_ON	IDLE	Ret	Ret
Power_OFF	OFF	OFF	OFF
Clocked_ON	Clocked	Clocked	Clocked
Block_Transfer	Active	Clocked	Active
Next_SDU	Active	Clocked	Clocked
Burst_Process	Active	Active	Active
Check_Write_Ops	Active	Active	Clocked
Wait_Pre_Burst	Clocked	Active	Clocked

Wait\_Pre\_Burst, Power\_ON, Power\_OFF, Block\_Transfer, Check\_Write\_Ops and Next\_SDU. For every functional state of a tile, its sub-components logic and memory could be in different power modes as shown in Table 4.1.

The performance model of the hardware accelerator has four ( $M=4$ ) tiles  $c_{1..M}$  and each tile  $c_i$  can take eight  $K_{c_i}=8$  discrete functional states  $S_{c_i}$ . The power model of the hardware accelerator in IDPA is called "coprocessor" and consists of twelve ( $Q=12$ ) components  $r_{1..Q}$ , and each component  $r_i$  can take four  $L_{r_i}=4$  different power states  $P_{r_i}$ . Thus every tile  $c_i$  in the performance model maps to three components  $r_{1,2,3}$  in the power model.

The value change dump file generated by the performance model, during the simulation of the use case, contains the information  $\mathbf{W}$  (see Equation 4.2) of functional states of each tile in a 32 bit vector in the binary form. An adapted value change dump file coming out of SystemC simulation of hardware accelerator performance model is shown in Listing 4.1.

How this information is decoded and mapped to the components in power model is shown in Figure 4.3 for  $\text{Tile}_0$  (i.e., the definition of  $F(\mathbf{W})$ , as introduced in Equation 4.4 above).

The hardware accelerator in the performance model is composed of 4 tiles. A 32-bit vector  $\text{Modem\_SoC.HW\_Accelerator.funcState}_n[31:0]$  from each tile is traced as shown in Figure 4.3. In this vector "n" corresponds to the tile number and "Modem\_SoC.HW\_Accelerator" depicts the hierarchy of the tile in the performance model.

Listing 4.2 shows a Java script object notation (JSON) file for  $\text{Tile}_0$  that acts as an interface and contains mapping information between performance and power model of hardware accelerator.

The *Mapping* in the middle is the most critical block in POEM, as it is developed from joint inputs of the performance and power models. A JSON file is used to make sure the power models can make use of the information coming out of the performance models in the form of value change dump file.

As the  $\text{Tile}_0$ 's power model is fine-grained, the  $\text{funcState}_0[31:0]$  is broken down into three parts, the first [0:3] bits contain the information for logic, the next [7:4] bits are for data memory and the last [11:8] bits are for instruction memory. Apart from that, a mapping between the different hierarchical naming is also required, "Coprocessor/Tile\_0/Logic@CurrentMode"

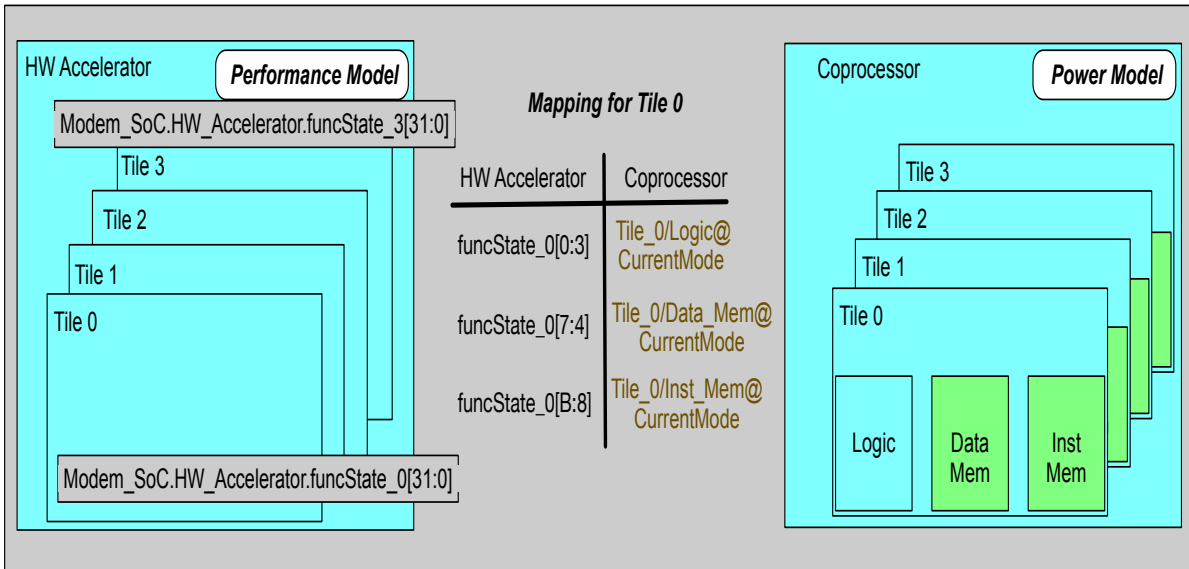


Figure 4.3: Mapping of the hardware accelerator’s stimuli (in the form of VCD) generated as a result of SystemC simulation of the performance model, on to the Coprocessor in the power model with the help of JSON file.

in the power model corresponds to the *Modem\_SoC.HW\_Accelerator.funcState\_0[3:0]* in the performance model.

```

1 $date
2     Jul 21, 2017      10:02:33
3 $end
4
5 $version
6 SystemC 2.3.1-Accellera --- Sep  2 2015 13:45:25
7 $end
8
9 $timescale
10     1 ps
11 $end
12
13 $scope module SystemC $end
14 $var wire    32  aaaaa Modem_SoC.HW_Accelerator.funcState_s_0 [31:0] $end
15 $var wire    32  aaaab Modem_SoC.HW_Accelerator.funcState_s_1 [31:0] $end
16 $var wire    32  aaaac Modem_SoC.HW_Accelerator.funcState_s_2 [31:0] $end
17 $var wire    32  aaaad Modem_SoC.HW_Accelerator.funcState_s_3 [31:0] $end
18 $var wire    32  aaaae Modem_SoC.HW_Accelerator.funcState_s_4 [31:0] $end
19
20 $upscope $end

```

```
21 $enddefinitions $end
22
23 $comment
24 All initial values are dumped below at time 0 sec = 0 timescale units.
25 $end
26
27 $dumpvars
28 b0 aaaaa
29 b0 aaaab
30 b0 aaaac
31 b0 aaaad
32 b0 aaaaee
33 $end
34
35 #100000000
36 b1100010011 aaaaa
37 b1100010011 aaaab
38 b1100010011 aaaac
39 b1100010011 aaaad
40 b1100010011 aaaaee
41
42
43 #100001000
44 b1000100010 aaaaa
45 b1000100010 aaaab
46 b1000100010 aaaac
47 b1000100010 aaaad
48 b1000100010 aaaaee
49
50
51 #100031248
52 b1100100010 aaaaa
53
54 #100059024
55 b1100100010 aaaac
56
57 #100086800
58 b1100100010 aaaad
59
60 #100197904
61 b1100110010 aaaaa
62
```

63 #102281104  
64 b1000110010 aaaaa  
65  
66 #102304048  
67 b1000100010 aaaaa  
68  
69 #102310992  
70 b1000110010 aaaaa  
71  
72 #102326992  
73 b1000100010 aaaaa  
74  
75 #102329104  
76 b1000110010 aaaaa  
77  
78 #102342992  
79 b1000100010 aaaaa  
80  
81 #102345104  
82 b1000110010 aaaaa  
83  
84 #102358992  
85 b1000100010 aaaaa  
86  
87 #102361104  
88 b1000110010 aaaaa  
89  
90 #102374992  
91 b1000100010 aaaaa  
92  
93 #102377104  
94 b1000110010 aaaaa  
95  
96 #102384048  
97 b1000100010 aaaaa  
98  
99 #102393104  
100 b1000110010 aaaaa  
101  
102 #102397936  
103 b1000100010 aaaaa  
104



```
105 #102404880
106 b1000110010 aaaaa
107
108 #102409104
109 b1100110010 aaaaa
110
111 $ ...
112
113 $ ...
114
115 $ ...
116
117
118 #4900000000
119 b100010001 aaaaa
120 b100010001 aaaab
121 b100010001 aaaac
122 b100010001 aaaad
123 b100010001 aaaae
124
125
126 #4900001000
127 b1000100010 aaaaa
128 b1000100010 aaaab
129 b1000100010 aaaac
130 b1000100010 aaaad
131 b1000100010 aaaae
```

Listing 4.1: A piece of VCD file generated as a result of SystemC simulation of the performance model of the hardware accelerator.

```
1 {"inputs": {
2     "Modem_SoC.HW_Accelerator.funcState_s_0_3_0":
3         {"params":["Coprocessor/Tile_0@CurrentMode"],
4          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'Idle
5          ', <input>==2, 'Clocked', <input>==3, 'Active')"],
6          "type": "int",
7          "default": 0},
8     "Modem_SoC.HW_Accelerator.funcState_s_0_7_4":
9         {"params":["Coprocessor/Tile_0_D@CurrentMode"],
10          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
11          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
12          "type": "int",
```

```

11         "default": 0},
12     "Modem_SoC.HW_Accelerator.funcState_s_0_B_8":
13         {"params":["Coprocessor/Tile_0_I@CurrentMode"],
14          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
15          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
16          "type": "int",
17          "default": 0},
18     "Modem_SoC.HW_Accelerator.funcState_s_1_3_0":
19         {"params":["Coprocessor/Tile_1@CurrentMode"],
20          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'Idle
21          ', <input>==2, 'Clocked', <input>==3, 'Active')"],
22          "type": "int",
23          "default": 0},
24     "Modem_SoC.HW_Accelerator.funcState_s_1_7_4":
25         {"params":["Coprocessor/Tile_1_D@CurrentMode"],
26          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
27          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
28          "type": "int",
29          "default": 0},
30     "Modem_SoC.HW_Accelerator.funcState_s_1_B_8":
31         {"params":["Coprocessor/Tile_1_I@CurrentMode"],
32          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
33          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
34          "type": "int",
35          "default": 0},
36     "Modem_SoC.HW_Accelerator.funcState_s_2_3_0":
37         {"params":["Coprocessor/Tile_2@CurrentMode"],
38          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'Idle
39          ', <input>==2, 'Clocked', <input>==3, 'Active')"],
40          "type": "int",
41          "default": 0},
42     "Modem_SoC.HW_Accelerator.funcState_s_2_7_4":
43         {"params":["Coprocessor/Tile_2_D@CurrentMode"],
44          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
45          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
46          "type": "int",
47          "default": 0},
48     "Modem_SoC.HW_Accelerator.funcState_s_2_B_8":
49         {"params":["Coprocessor/Tile_2_I@CurrentMode"],
50          "values":["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
51          <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
52          "type": "int",
53          "default": 0}

```

```
46         "type": "int",
47         "default": 0},
48     "Modem_SoC.HW_Accelerator.funcState_s_3_3_0":
49         {"params": ["Coprocessor/Tile_3@CurrentMode"],
50         "values": ["builtins.IF(<input>==0, 'OFF',<input>==1, 'Idle
51 ', <input>==2, 'Clocked', <input>==3, 'Active')"],
52         "type": "int",
53         "default": 0},
54     "Modem_SoC.HW_Accelerator.funcState_s_3_7_4":
55         {"params": ["Coprocessor/Tile_3_D@CurrentMode"],
56         "values": ["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
57 <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
58         "type": "int",
59         "default": 0},
60     "Modem_SoC.HW_Accelerator.funcState_s_3_B_8":
61         {"params": ["Coprocessor/Tile_3_I@CurrentMode"],
62         "values": ["builtins.IF(<input>==0, 'OFF',<input>==1, 'RET',
63 <input>==2, 'Clocked', <input>==3, 'ACTIVE')"],
64         "type": "int",
65     }
66 }
```

Listing 4.2: JSON file for mapping of the hardware accelerator's stimuli generated as a result of SystemC simulation of the performance model, on to the Co-processor.

## 5 Network Processor Case Study

In order to validate and showcase the effectiveness of the proposed methodology, an architectural exploration of a network processor for performance and power is presented in this chapter. The idea of the network processor case study is taken from the work reported in TAPES [56]. The difference lies in the approach used for performance analysis in TAPES and other extra features evaluated for packet routing in this dissertation.

We start with evaluating the application with stand-alone simulation using virtual engines. Virtual engines are developed in SystemC and Accellera library on top, discussed in 3.2.2 in detail already. After refining the application with virtual engines' help, co-simulation of the application and architectural model gives an insight into the resources required to get the optimal performance key performance indicators. We conclude this chapter with a network processor case study for power and performance co-optimization for a given set of constraints. Co-optimization at run time (software modification) is carried out by changing the parameters like frequency, packet size, and mapping file in our example. And based on the simulation results, the optimizations like memory size, memory word size, number of cores are done (in hardware configurations) for the next iteration until we achieve our target key performance indicators. POEM is used to compare different hardware configurations for power and performance with a systematic and modular framework.

### 5.1 System level uses of Mobile Communication Platforms

The outcome and contribution of this work are primarily related to the development of 5G modems at Intel® Deutschland; However, due to confidentiality reasons, we are explaining the relevance now with a general network processor example. As this work is done in collaboration with Intel's Mobile Communications Division, the focus was on protocol stack development of modems. In Intel®, this smart engine was developed in the first place to achieve cross-layer optimizations in the terminal. This smart engine is very similar to the hardware accelerator we have discussed in 4.1.4.

There are low-envelope power use cases that are critical for cellular modem architecture design perspective and dictate its design. These use cases belong to Connected Mode DRX (CDRX). Such use cases have traffic patterns and data transfer on the uplink, and downlink happens in bursts. An inactivity period between data bursts is crucial from a power perspective as the cellular modem can be in low power mode or even in power gated state. The proposed methodology contributed significantly to identifying such periods and doing the non-trivial trade-off analysis of save and restore power penalty (required in incase of power gating a subsystem) and power saved due to power gating. POEM was successfully

applied in making system level decisions like power domain division, hardware-software split, micro-architecture path findings, on-chip off-chip memory sizing, and implementing left shift strategy for power and performance verification.

## 5.2 Network Processor Use Case

The basic functionality of a network processor is to receive packets at a certain rate from its input ports, process them and route them to its output ports. A packet header is processed only to determine the destination port without performing any packet classification or security checks in TAPES. However, according to [94], the demand for high quality of service of current network systems requires giving some kinds of packets higher priority than others, and also security checks need to be applied on packets for safe communication.

In this case-study, we analyze both performance and power of a network processor, and in addition to packet routing, we also consider the load for packet classification and security check. The security check in our use case means applying a firewall (state-full firewall) for allowing packets only for known active connections and blocking the others. Same as in TAPES, we assume that all the packets have 20 bytes header and output port distribution for all the received packets are equal. Additionally, we assume that the network processor only processes the 20 byte packet headers, and the security check result is always non-blocking (all the packets are allowed to pass through). The number of instructions necessary for executing each task of this application, on a 32-bit processor, is assumed to be as in Table 5.1.

Table 5.1: Number of instructions of a network processor application tasks with CPI=1.2

<i>Task Name</i>	cost(#instructions)	#CPU cycles(For CPI = 1.2)
<i>Packet routing</i>	400	480
<i>Packet classification</i>	1000	1200
<i>Stateful firewall</i>	3000	3600

As a first design step, we evaluate the application itself by performing a stand-alone simulation using only the virtual engines. After analyzing the results of this step, an initial architecture and mapping configuration is determined. Then we set up the corresponding initial architecture using the SAVE framework (see details in 3.2) performance models, evaluate its feasibility, and improve it step by step. First the performance of the application for a pure software based solution is evaluated, where all the processing and packet management is done inside the CPU. Then some of the CPU tasks are offloaded on a hardware accelerator and DMA channels.

The flow of a typical network processor application is as follow:

Step-1 The received packets are stored in the memory by a buffer manager and a descriptor to each packet. The descriptor shows where the packet resides in the memory is stored in an input-queue.

Step-2 The stored packets are read from memory according to their descriptors in the input-queue. Each packet is classified, checked for security, and its destination port is specified. The processed packets are written back to the memory, and the descriptor of the corresponding packets are sent to the output queue manager.

Step-3 The processed packets are retrieved from the memory, according to their descriptors in the output queue, by the queue manager and sent to their corresponding output ports.

Table 5.2: Network processor specifications

Components	Specifications
Input port1	100 Mbps Ethernet ports
Input port2	
Output port1	
Output port2	
Memory block	Off-chip single port 8 or 16 bit SDRAM
Processing cores	Flexible

The network processor has two 100 Mbps Ethernet input ports and two output ports, shown in table 5.2. Unlike TAPES, we model the application as a task graph having seven tasks, shown in Figure 5.1. The tasks, *Read\_1* and *Read\_2*, read the received packets from the two input ports. Then the packets are classified, firewall check is applied, output ports' routing information is added, by the *Classify*, *Firewall* and *Route* tasks, respectively. The *Transmit\_1* and *Transmit\_2* tasks write the processed packets to the two output ports, respectively. It is assumed that, after arriving in the input ports, each packet is written to the memory. Therefore, tasks *Read\_1* and *Read\_2* are configured as periodic tasks, and their period represents the inter-arrival time of the packets. This period is configured for each packet size and input data rate separately. The triggering mode of the task *Classify* is configured as `OR_MODE`. Therefore, it is triggered whenever there is data available in any of its input channels. When both of its input channels have data simultaneously, then it reads them in a round-robin fashion. Similarly, task *Route*'s output triggering mode is also configured as `OR_MODE`. Upon completion, it triggers only one of its successor tasks. The sequence of triggering its output task is configured in a round-robin fashion to ensure uniform packet distribution in the output ports.

### 5.3 Stand-Alone Application Simulation Using Virtual Engines

In this section, the application itself is simulated using virtual engines. The workload of each task on a virtual engine, as stated in Section 3.1.1, has only one parameter: the corresponding task's processing time. The workloads for this network processing application's tasks are approximated based on the frequency and number of instructions for executing the corresponding tasks. Since the virtual-engines do not generate any traffic, the interconnect and memory subsystem, latency, and contentions, are not considered at this stage.

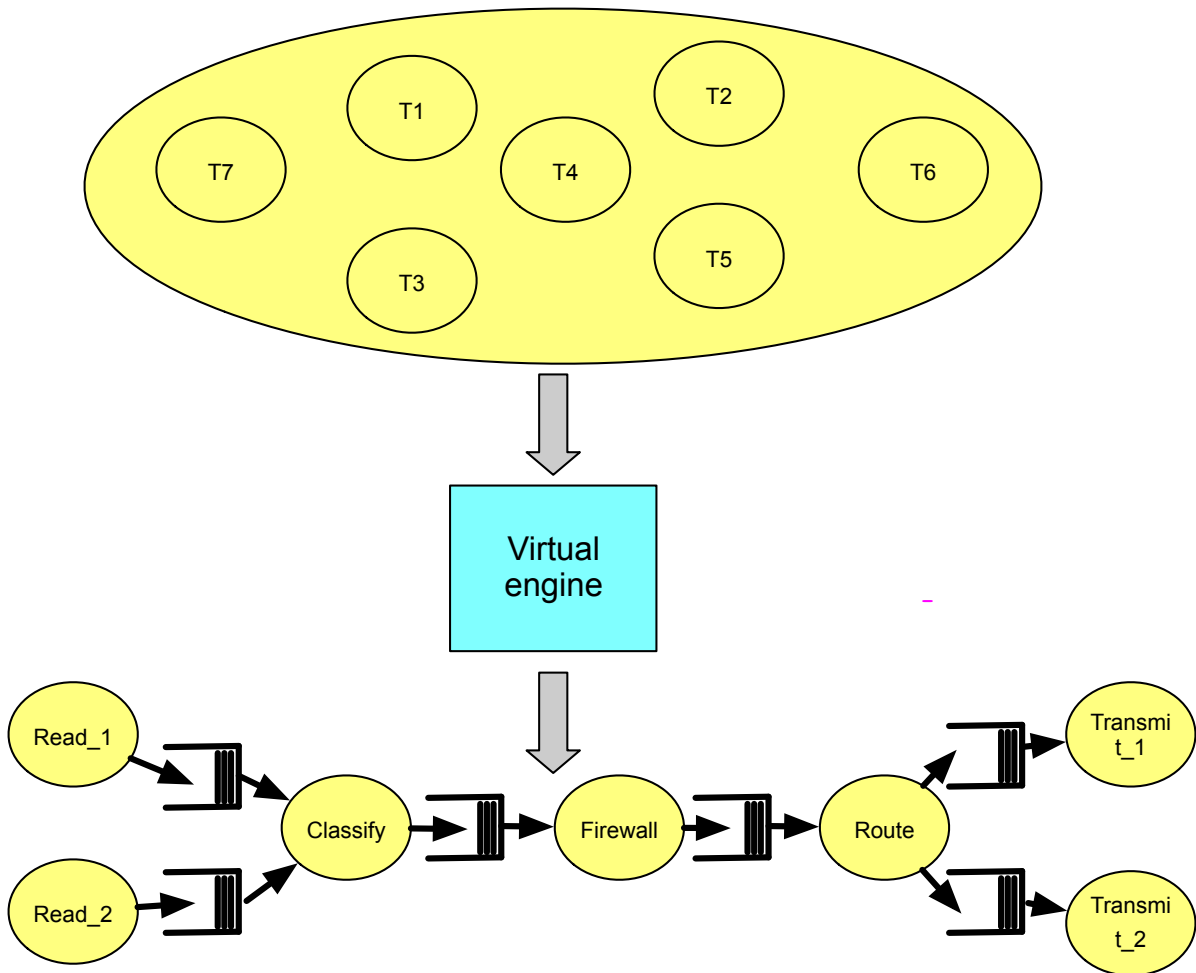


Figure 5.1: Network application task graph sequence shown with seven task graph nodes and FIFO channels in between them after analyzing with a virtual engine.

### 5.3.1 Mapping All Tasks on a Single Virtual Engine

At first, a pure-software based solution is evaluated, where all of the 7 tasks are mapped on a single virtual-engine. The virtual-engine at this stage serves as a meta-model for a 32 bit CPU running at 500 MHz, shown in table 5.4. The workload of each task on a virtual engine for 500 MHz frequency, is shown in the table 5.3.

After performing an initial simulation, it was observed that the engine was mostly occupied by the *Read\_1*, *Read\_2*, and *Classify* tasks. Therefore, tasks' priorities were set in increasing order from left to right, and further simulations were carried out. Figure 5.2a shows the throughput or aggregate output data rate, in a million bits per second (Mbps), of the network processor as a function of its input data rate. It shows that the throughput for shorter packets is lower than the throughput for the larger ones. For 64-byte packets, the throughput saturates at 50 Mbps, but for 256-byte packets, it saturates at 190 Mbps. This saturation behavior is

Table 5.3: Virtual engine workload load for different tasks

Task name	Virtual-engine workload (ns)
<i>Packet routing</i>	960
<i>Packet classification</i>	2400
<i>Firewall check</i>	7200
<i>Read_1</i>	This value depends on the size of the packet. For a 64 byte packet it is 77 ns
<i>Read_2</i>	
<i>Transmit_1</i>	
<i>Transmit_2</i>	

Table 5.4: Use case modeled in the form of task graphs mapped on to single virtual engine

Task name	Processing element
<i>Packet routing</i>	Virtual engine
<i>Packet classification</i>	
<i>Firewall check</i>	
<i>Read_1</i>	
<i>Read_2</i>	
<i>Transmit_1</i>	
<i>Transmit_2</i>	

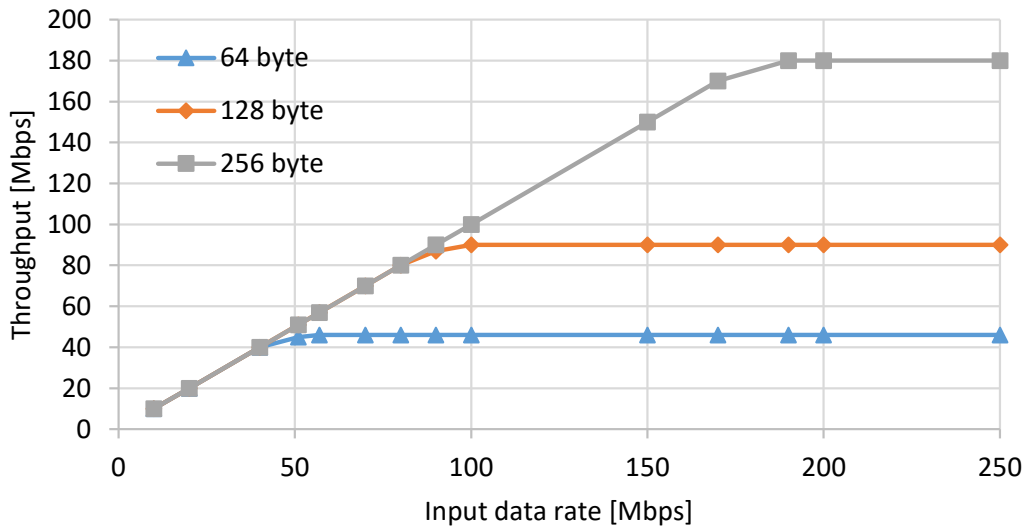
because of processing only the 20-byte header of all kinds of packets, either short or long. Consequently, for the same input data rate, a large number of shorter packets enter the network processor than longer ones. Therefore, in the case of 64-byte packets, the network processor has to process many 20-byte headers than for the 128 and 256-byte packets.

However, the maximum throughput with respect to the number of packets processed per second, as shown in Figure 5.2b, is slightly higher for the smaller packets than the larger ones. The lower throughput for larger packets is because considering the latency increase while writing them to memory when received and reading them from memory for transmitting to the output ports when processed.

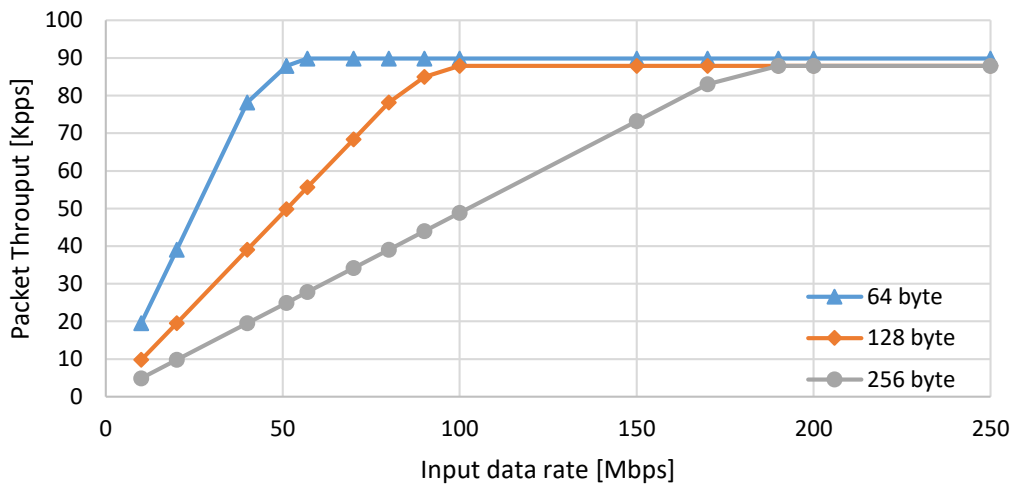
Similarly, Figure 5.3 shows the virtual engine's average usage or load imposed by each task, while processing a total number of 800 packets in case of maximum throughput. It shows that the *Firewall* task mainly occupies the engine.

These simulation results show that the pure software based solution using one 32-bit CPU core of 500 MHz frequency cannot handle the processing of packets from two 100 Mbps input ports. One logical option to improve the system's performance is to map the heaviest task *Firewall* on a separate processing element. Next, we want to see the throughput improvement when the *Firewall* task is running on a separate virtual engine with the same speed as virtual engine used in this section. Additionally, the tasks *Read\_1*, *Read\_2*, *Transmit\_1*, and *Transmit\_2* are offloaded to another virtual engine.





(a)



(b)

Figure 5.2: System's corresponding data throughput (a) and packet throughput (b) (consisting of a single virtual engine) at different input data rates and packet sizes.

### 5.3.2 Mapping Tasks on Multiple Virtual Engines

In this section, the mapping configuration shown in table 5.5 is evaluated. In this configuration, the *Firewall* task is mapped on a separate virtual engine. *Classify*, and *Route* tasks share one virtual engine, and similarly, *Read<sub>x</sub>* and *Transmit<sub>x</sub>* tasks share another virtual engine. The execution speed and workload of all three virtual engine tasks are the same as those used in the previous section 5.3.1. It can be seen from the simulation results of this configuration

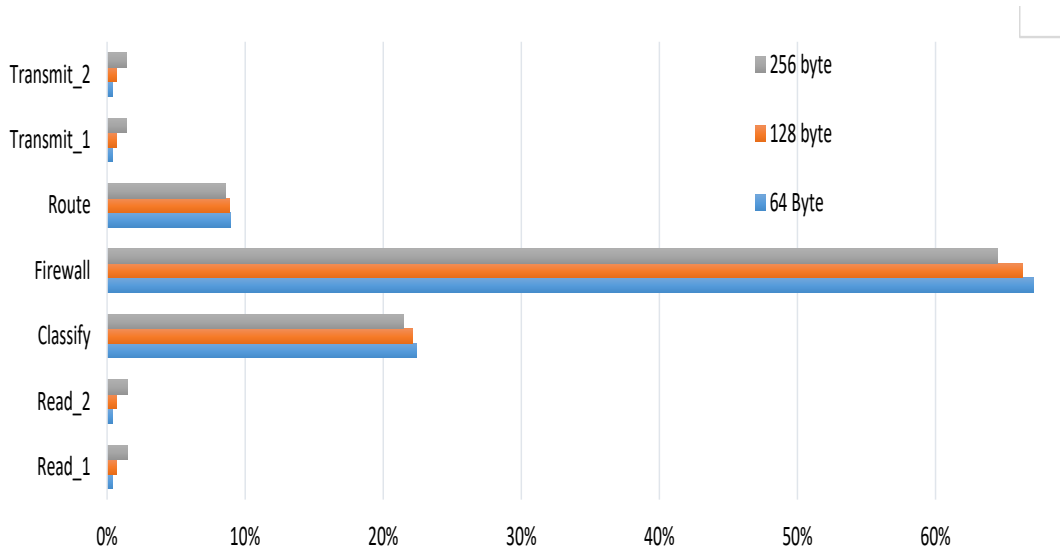


Figure 5.3: Virtual engine usage by each task for maximum throughput. Firewall Classify and Route are the three tasks that contribute towards the virtual engine usage, respectively, for all three packet sizes.

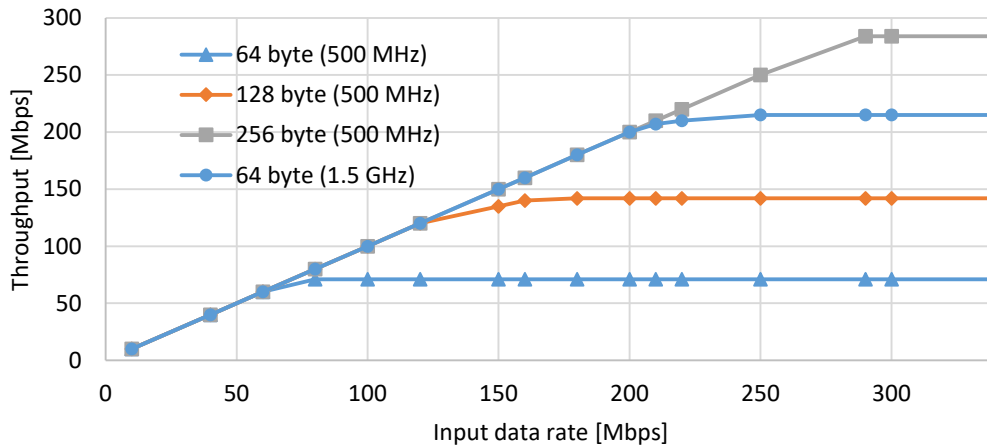
(shown in Figure 5.4), that there is a 50% output data throughput increase for all packet sizes as compared to the single virtual engine configuration.

The packet throughput, shown in Figure 5.4b, has also improved with the same scale, but it saturates around 140 Kbps for all three types of packets. This throughput was slightly higher for 64-byte packets than the others in the previous configuration, using one virtual engine. Throughput is equal for all packet sizes in the multiple virtual engine case because the tasks *Read\_x*, and *Transmit\_x* are offloaded from the packet processing engines. Therefore, the virtual engine-2 and virtual engine-3's processing load is the same for long and short packets.

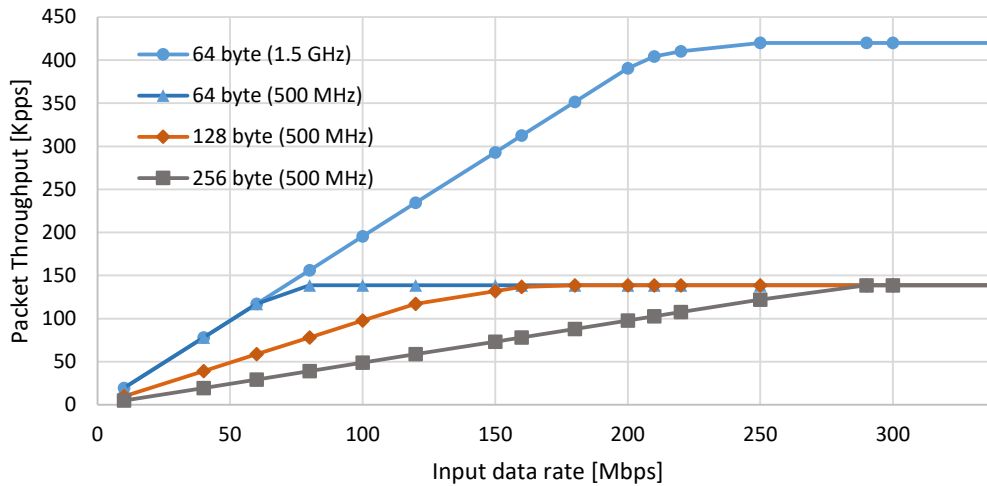
Table 5.5: Use case modeled in the form of task graphs mapped on to 3 virtual engines

Task name	Processing element
<i>Packet routing</i>	Virtual engine 3
<i>Packet classification</i>	
<i>Firewall check</i>	Virtual engine 2
<i>Read_1</i>	Virtual engine 1
<i>Read_2</i>	
<i>Transmit_1</i>	
<i>Transmit_2</i>	

Simulation results in Figure 5.4a show that using three processing elements, the throughput for 256-byte packets is above 200 Mbps, but for 128 and 64-byte packets, it is still not enough for handling two 100 Mbps Ethernet ports. Figure 5.5 shows the average execution latency of



(a)

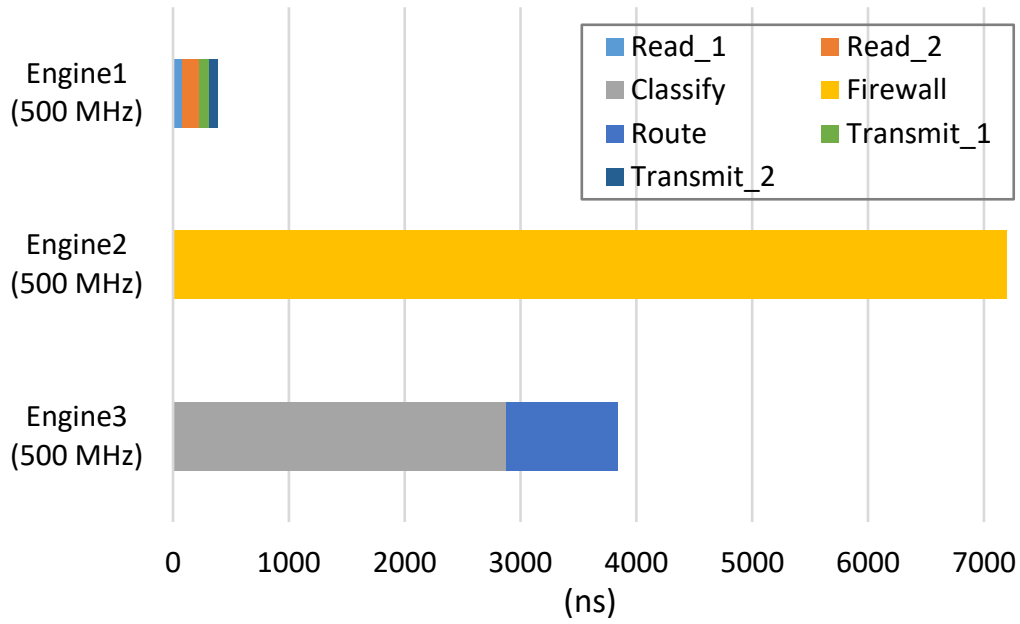


(b)

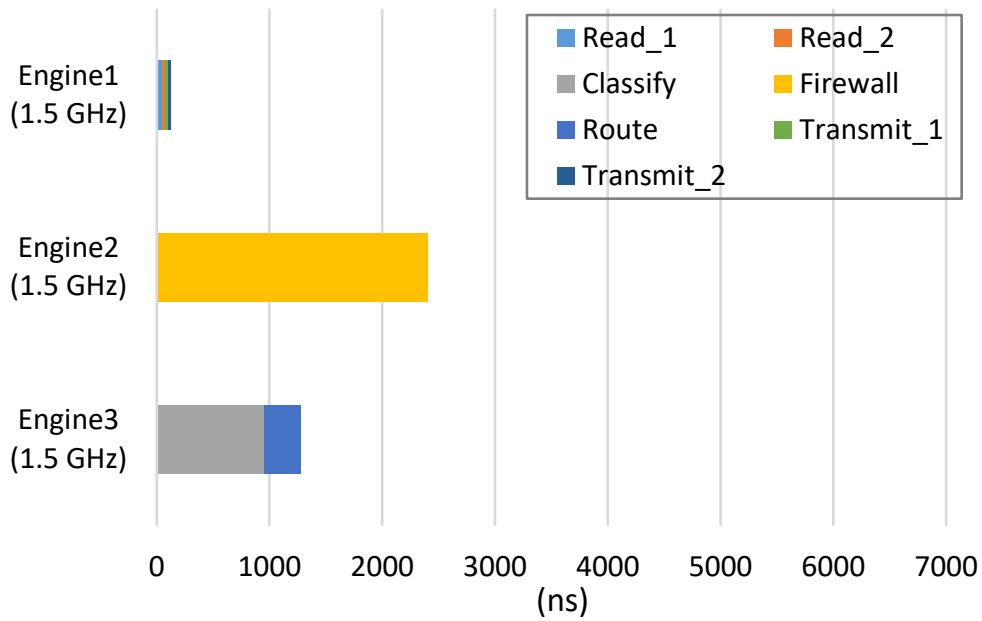
Figure 5.4: System's corresponding data throughput (a) and packet throughput (b) (consisting of multiple virtual engines) at different input data rates and packet sizes.

the tasks on their corresponding virtual engines. It can be seen that the execution latency of the *Firewall* task, mapped on Engine2, is relatively longer than the other tasks and it affects the system throughput. The frequency of the processing elements is increased to 1500 MHz to decrease this task's execution latency. It can be seen from the simulation results shown in Figure 5.4a that the throughput of 215 Mbps can be achieved for 64-byte packets, and throughput of 128 and 256-byte packets will be higher than 64-byte packets for this new frequency.

Figure 5.5 also shows that the execution latency of the tasks (mapped on Engine1), responsible for writing received packet to the memory and reading the processed packets from



(a)



(b)

Figure 5.5: Average execution latency of the tasks on the corresponding engines for 500 MHz (a) and 1.5 GHz (b).

the memory, is very short. This low latency is because while using the virtual engines, the interconnect and memory subsystem latency and contentions are not considered. In the

following section performance of this application for an actual architectural configuration is examined. And the virtual engines are replaced with more detailed performance models. While executing a task, these performance models access the memory block via the interconnect and accordingly generate traffic. Therefore, it is possible to observe the contentions and load of the interconnect and the memory subsystems.

## 5.4 Co-simulation of the Application and the Architecture Model

The application's performance on an architecture model, consisting of processing elements, interconnect, and memory blocks, is evaluated in this section. As an initial decision for the number and configuration of the processing cores and the mapping configuration of the tasks, the previous section's results are used.

### 5.4.1 Mapping Tasks on an Architecture Model With Two CPU Cores

In stand-alone application simulation using virtual engines, it was observed that for obtaining a throughput of 200 Mbps, assuming no memory and interconnect contentions, a minimum of two CPU cores running at 1.5 GHz is needed. Therefore, in this section, we first evaluate a network processor architecture's performance, shown in table 5.6. This architecture consists of two 1.5 GHz CPU cores (replacing engine-2 and engine-3), a DMA model (replacing engine-1), an interconnect, and an off-chip memory block. direct memory access (DMA) is a component in an embedded system used to efficiently transfer data from one memory location to another without involving the CPU.

Table 5.6: Use case modeled in the form of task graphs mapped on to network processor performance model with 2 CPU cores and a DMA

<i>Task name</i>	Processing element
<i>Packet routing</i>	Core 1
<i>Packet classification</i>	
<i>Firewall check</i>	Core 2
<i>Read_1</i>	DMA
<i>Read_2</i>	
<i>Transmit_1</i>	
<i>Transmit_2</i>	

The CPU cores' workload objects for each task are configured according to the values presented in Table 5.1. Further, it is assumed that each CPU core has its private data and program caches of size at least one network packet (64, 128, or 256 bytes) and the program instructions, respectively. The first instruction for executing each task is considered data cache-miss, resulting in a memory transaction of burst-size 4 bytes and length 5, representing a 20-byte packet header reading from memory. Similarly, when the reading transaction is completed, the header's processing time is defined by the frequency and number of instructions (table 5.1) required for each task. Finally, writing back the modified packet

header to the memory is also carried out through a transaction of burst-size 4 bytes and length 5.

The memory block is configured as an off-chip SDRAM of word-depth 1k and word-width 8 bits running at 100 MHz. SDRAM read and write access latencies are configured as 15 and 10 clock cycles, respectively. Each read or write transaction request is served as bursts of size 8 and length 4. Each beat's reading, representing a word of size 8 bits, within a burst transaction is configured as two clock cycles and, similarly, writing each beat as one clock cycle.

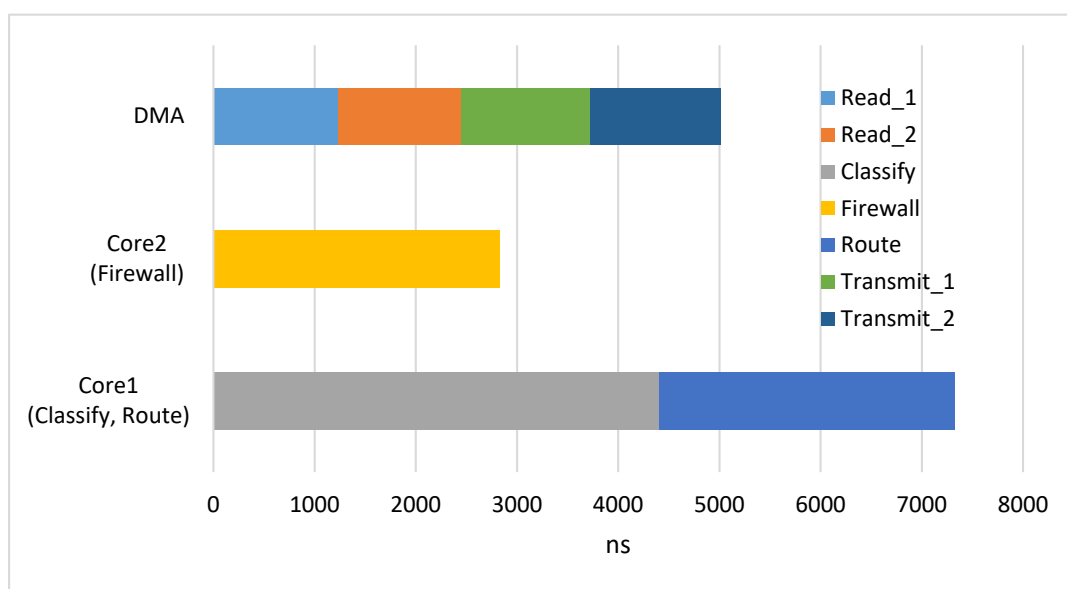
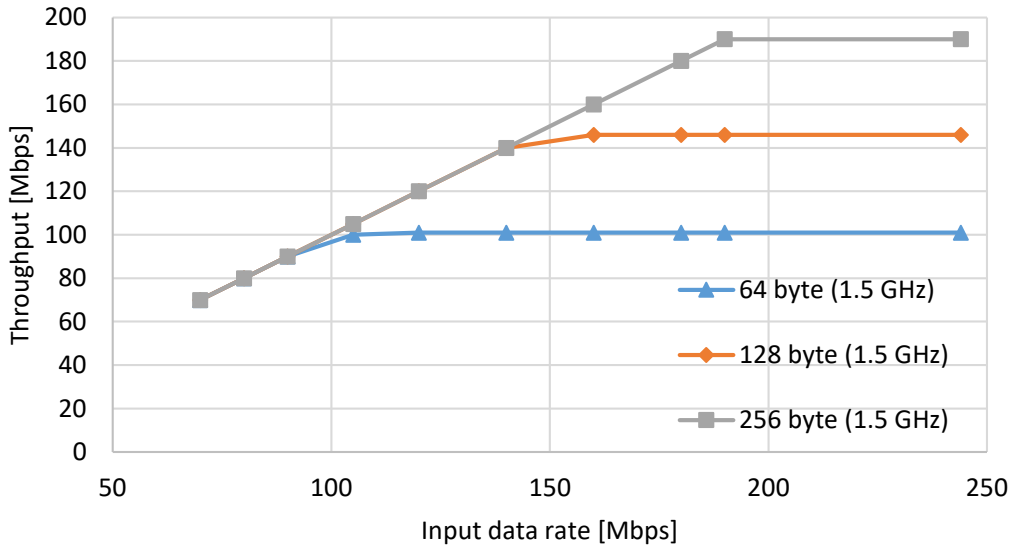


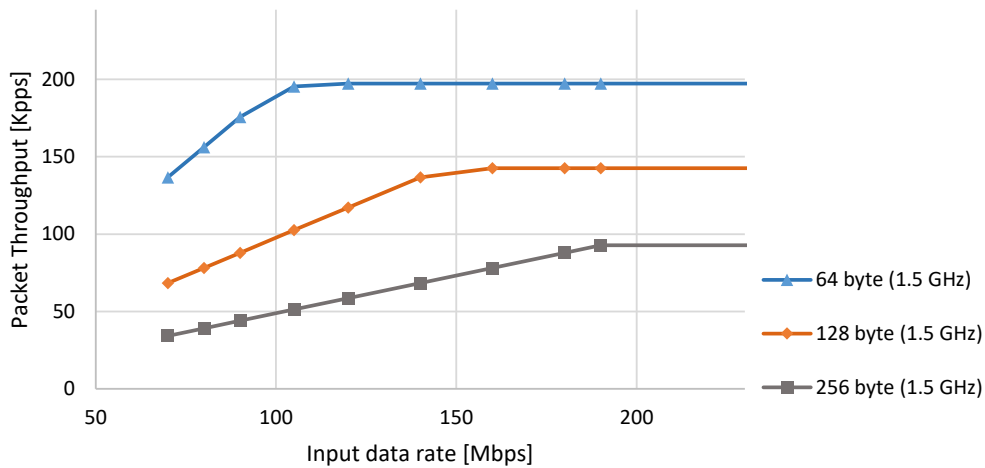
Figure 5.6: Average execution latency of the tasks on the corresponding engines for 2 CPU core (1.5 GHz) configuration.

The NoC is configured as a crossbar with round-robin arbitration. The DMA channels are modeled as a traffic generator, which generates traffic according to the workload of the tasks mapped on to it. For instance, in table 5.6 *Read\_1* and *Read\_2* are mapped on the DMA module; therefore, they periodically trigger this traffic generator to generate write transaction requests of packet-size on the NoC for writing the received packets to the memory block. Similarly, *Transmit\_1* and *Transmit\_2* are also mapped on this traffic generator, which requests similar transactions of a read request. The workload object of *Read\_x* and *Transmit\_x* tasks only differ in read and write commands.

The above stated architecture and task graph with corresponding mapping relations are simulated. The simulation results in Figure 5.7a shows that the throughput of the system for 64-byte packet length, unlike the results in Figure 5.4a, is below 200 MHz. Though, the number and frequency of the cores are the same as those used in Section 5.3.2. Similarly, Figure 5.7b shows that shorter packets have larger packet throughput than longer ones, unlike the packet throughput in Figure 5.4b. These differences in performance are the memory and interconnect latencies and contentions, which are not considered in the stand-alone simulation



(a)



(b)

Figure 5.7: System’s performance for 2 CPU cores (1.5 GHz) a) data throughput, b) packet throughput.

using virtual engines.

The effects of these memory latencies and contentions can be seen by comparing tasks’, *Read\_1*, *Read\_2*, *Transmit\_1*, and *Transmit\_2* average execution latencies in Figure 5.6 with Figure 5.5b. There is a 2500% increase in these tasks’ execution latencies, which are mapped on to the Engine1 in stand-alone simulation and on the DMA module in this section.

By inspecting the average execution latencies of the tasks for 64-byte packets, shown in

Figure 5.6, one can observe that the bottleneck of this design configuration is the performance of the core1, on which the tasks *Classify* and *Route* are mapped. Therefore, in order to improve the system's performance, the number of cores is increased to 3, and the task *Route* is mapped on this third core, which is also running at 1.5 GHz. Without changing the other configurations of the system and mapping relations of the other tasks, we evaluate the three core architecture's performance in the next section.

#### 5.4.2 Mapping Tasks on an Architecture Model With Three CPU Cores

In this section, the architecture's performance with three CPU cores and the tasks' mapping relations, as shown in table 5.7, is evaluated. All three cores are running at 1.5 GHz, and the memory and DMA configurations are the same as in the previous section 5.4.1.

Figure's 5.8 simulation results show that there is only around 14% improvement in the system's maximum throughput. The throughput of the 64-byte length packets saturates at around 115 Mbps, which is not enough for handling two 100 Mbps input ports of the network processor. Similarly, Figure 5.9 shows the average execution latency of the tasks. It can be seen that the execution latency of the tasks mapped on the DMA module, whose functions are writing the received packets to the memory and reading the processed packets from memory, are now the bottleneck of the design. These tasks' execution latency is mainly caused by the contentions and latencies of the memory block and the NoC.

Table 5.7: Use case modeled in the form of task graphs mapped on to network processor performance model with 3 CPU cores and a DMA

Task name	Processing element
<i>Packet routing</i>	Core 3
<i>Packet classification</i>	Core 1
<i>Firewall check</i>	Core 2
<i>Read_1</i>	DMA
<i>Read_2</i>	
<i>Transmit_1</i>	
<i>Transmit_2</i>	

#### 5.4.3 Increasing the Memory Size and Result Analysis

In the previous section, after analyzing the simulation results, it was observed that the latencies and contentions of the memory block and the interconnect cause the increase in average execution latency of the tasks *Read\_x* and *Transmit\_x*. These tasks are creating the bottleneck for gaining a throughput of above 200 Mbps for 64-byte packet lengths. Therefore, in this section the performance of the system is evaluated with an improved memory block. The SDRAM memory block's width is increased from 8 bits to 16 bits, and its frequency is increased from 100 MHz to 133 MHz.

Figure's 5.10 simulation results show that the DMA tasks' average execution latency has improved by more than 50%. Similarly, Figure 5.11a, shows that the maximum throughput



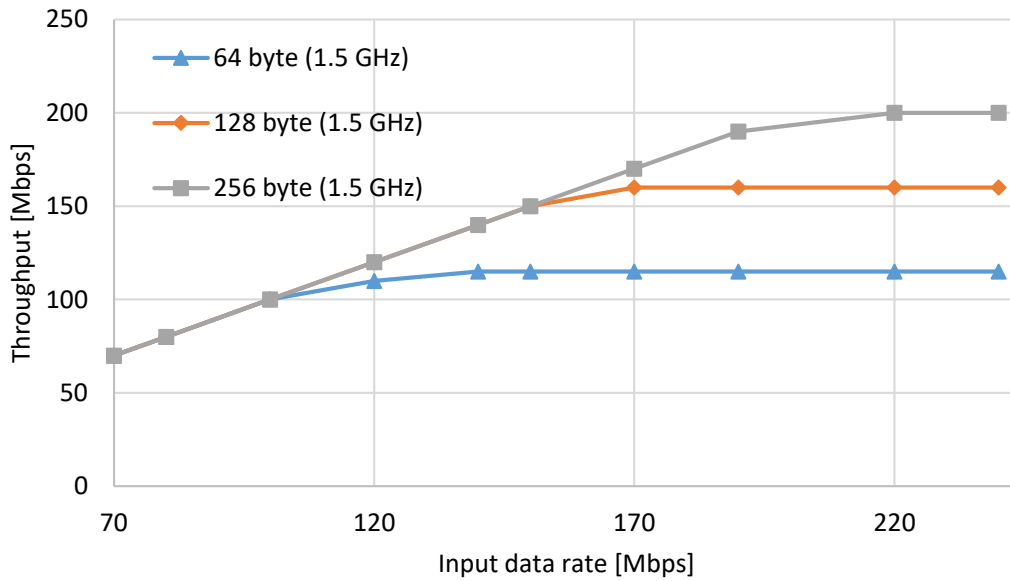


Figure 5.8: System’s performance for 3 CPU cores at 1.5 GHz frequency at three different packet sizes, the system run with 256-bytes packet size can archive max output throughput.

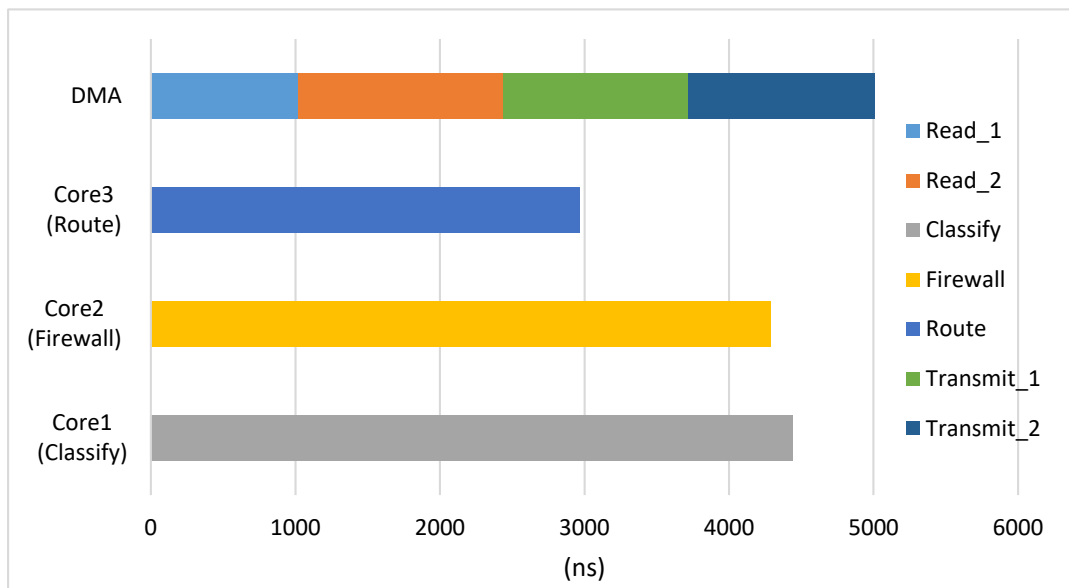


Figure 5.9: Average execution latency of the tasks on their corresponding engines for 3 CPU core (1.5 GHz) configuration.

for the 64-bytes packets, is above 200 Mbps, which is adequate for serving two 100 Mbps input Ethernet ports of the network processor.

At this point, we have derived the network processor (architecture) configuration, system

parameters (i.e., frequency of processing elements) from a step-by-setup refinement of the application and architecture model. This activity corresponded to Layer-1 of our methodology POEM, as shown in Figure 4.1 and discussed in Section 4.1.

All the implementation changes, i.e., mapping tasks on different cores (modification of application model) and increasing the number of cores (modification in the architecture model) to mitigate the resource contention, do not affect each other. As an outcome, the achieved modularity and separation-of-concern back the claim of POEM. Moreover, one run from the other can be done on the fly by modifying the configuration file or the system architecture validation environment (SAVE) environment's initiation file. SAVE is an Intel internal environment that consists of SystemC libraries and focuses on transaction modeling at an approximate abstraction level see details in Section 3.1.2.

The design space exploration emphasis was on output throughput (performance) for the network processor. Next, we will re-evaluate this network processor configuration by building on top of the derived configuration to co-optimize power and performance. The CPU's performance model will be replaced with an hardware accelerator (as discussed in Section 4.1.4). The stimuli coming out of the SystemC simulation of the configuration will be captured in value change dump (VCD) file. Later to be used in Intel® Docea™ for power analysis.

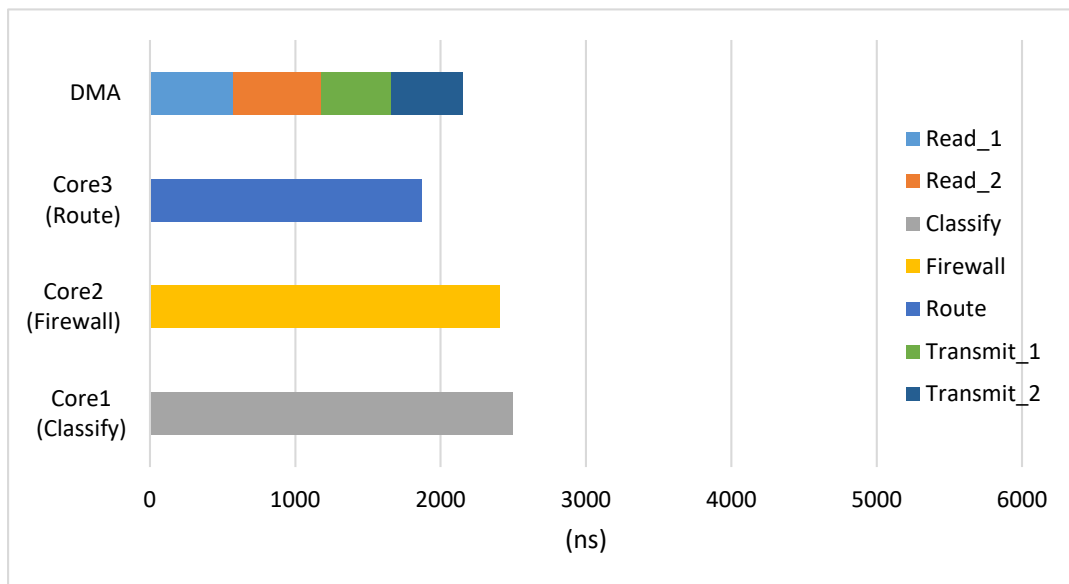
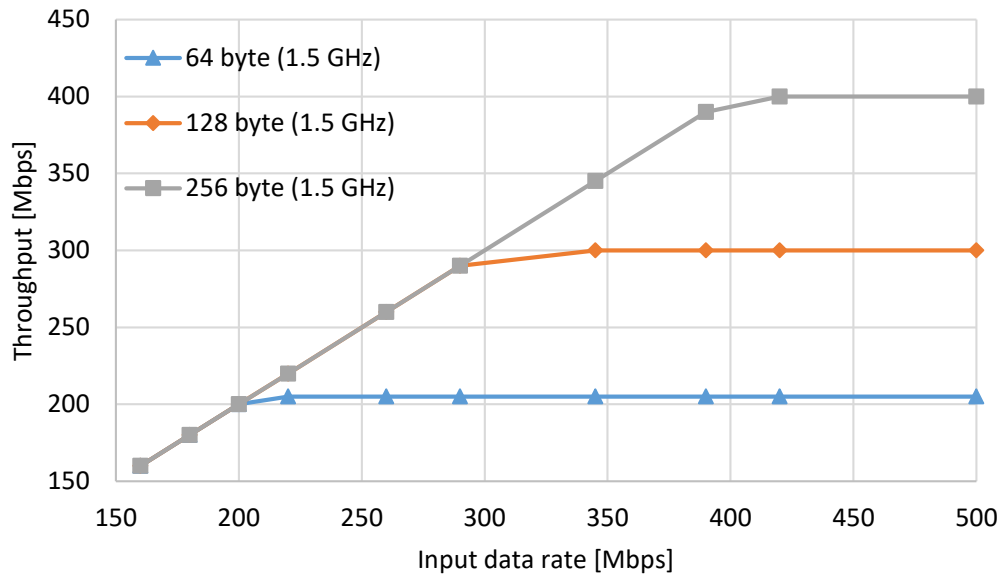


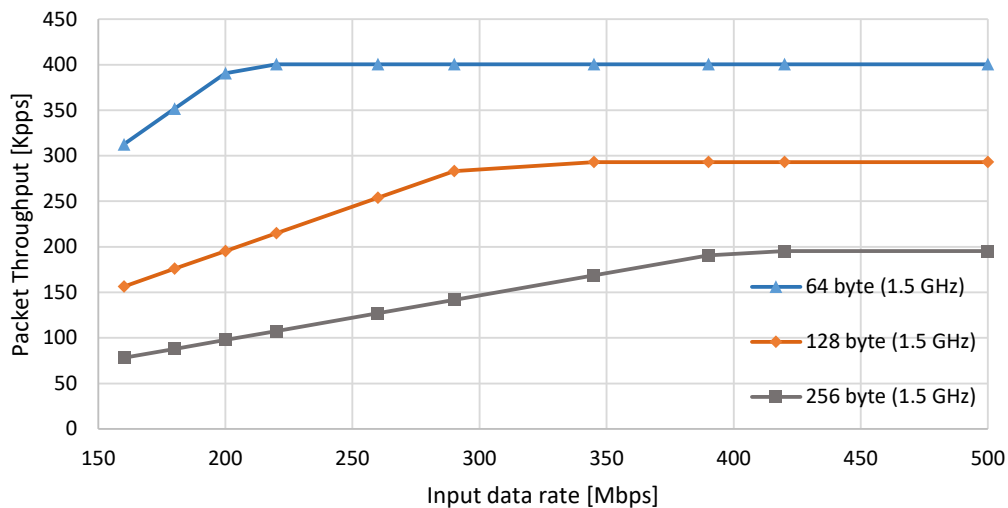
Figure 5.10: Average execution latency of the tasks on their corresponding engines for 3 CPU core (1.5 GHz) with improved memory block.

## 5.5 Power and Performance Co-Simulation

Co-optimization of power and performance for multiprocessor system on chip, i.e., the cellular modem, is inevitable (as reported in [5]); thus, one of the motivations behind this research



(a)



(b)

Figure 5.11: System performance for 3 CPU cores (1.5 GHz) with improved memory block.

work. POEM works towards this goal by keeping the application, architecture, and power modeling setup modular.

The connection between SystemC simulation (at TLM) and power modeling framework is offline with the help of a value change dump file, which contains the state-residencies of the performance model's processing elements ,i.e., cores, memory, buses, and hardware accelerators. By state-residence, we mean the discrete power state (that is characterized and back annotated based on its functional state in the first place) and time spent in this power

state by a component. The interface and mapping between the SystemC simulation of the performance model and the power modeling framework have been explained (see details in Section 4.1.4).

We will start with the three CPU core configuration, which we derived in Section 5.4.2, with slight modifications for co-optimization and power and performance analysis. In the process, we will explore system parameters and play with the application to achieve the desired results.

The performance model of this network processor consists of a network on chip (NoC), memory, direct memory access (DMA) controller, and a 32-bit 3-core/tile CPU. Hardware accelerator explained in 4.1.4 will be used here as a CPU. Tasks *Read\_1*, *Read\_2*, *Transmit\_1* and *Transmit\_2* are mapped on DMA. *Classify*, *Firewall* and *Route* can be mapped on the three tiles of the CPU, It is assumed that the CPU has an L1 cache, which can access only 3 network packets at a time.

### 5.5.1 Simulation Environment Settings and Goals

The simulation parameters fixed for this use case are voltage  $V_{dd}$  equal to 0.9 V, and simulated run-time  $t$  equal to 4 ms. In this case study for a fixed input throughput of 200 Mbps, the goal is to achieve an output (max) throughput of 200 Mbps in an optimal way. Similarly, the power dissipation calculated here is an average power that includes dynamic and static power.

The application modeled is mapped to fixed architecture resource in two potential ways, called configurations, as shown in Figure 5.12 and Figure. 5.13. *Configuration\_1* and *Configuration\_2* are simulated for 64-byte packet size over a frequency range, in order to see how mapping the *Firewall* task to two tiles (*Tile\_1* and *Tile\_2*) and task of *Classify* and *Route* to *Tile\_0* impact the system w.r.t. power and performance, as compared to the first *Configuration\_1*.

The results for throughput and power are shown in Figure 5.14a and Figure 5.14b respectively, and they clearly indicate that *Configuration\_2* can achieve a higher throughput than *Configuration\_1* but, that higher throughput comes at the cost of higher power dissipation as well. The reason behind is that, as the *Firewall* needs the double number of the clock cycles as compared to *Classify* and *Route* tasks together (as shown in Table 5.1), it is a bottleneck for higher throughput in *Configuration\_1*. Hence, mapping it to two different tiles will increase the throughput.

It is important to note that with a 64-byte packet size, both configurations fail to achieve even a 100 Mbps throughput. Therefore, as a next step, we simulate *Configuration\_1* for different packet sizes and increase the frequency range even further to achieve 200 Mbps throughput. *Configuration\_2* is not persuaded going forward in this work. Although we can achieve higher throughput with this configuration, it comes at the cost of almost double the power dissipation.

Results shown in Figure 5.15a, highlight that *Configuration\_1* achieves max throughput for 128 and 256-byte packet size at 500 and 833.33 MHz frequency respectively, but it fails to achieve a 150 Mbps throughput for a 64-byte packet size for frequency as high as 1 GHz. This is because the network processor only processes packet headers that are fixed for both large

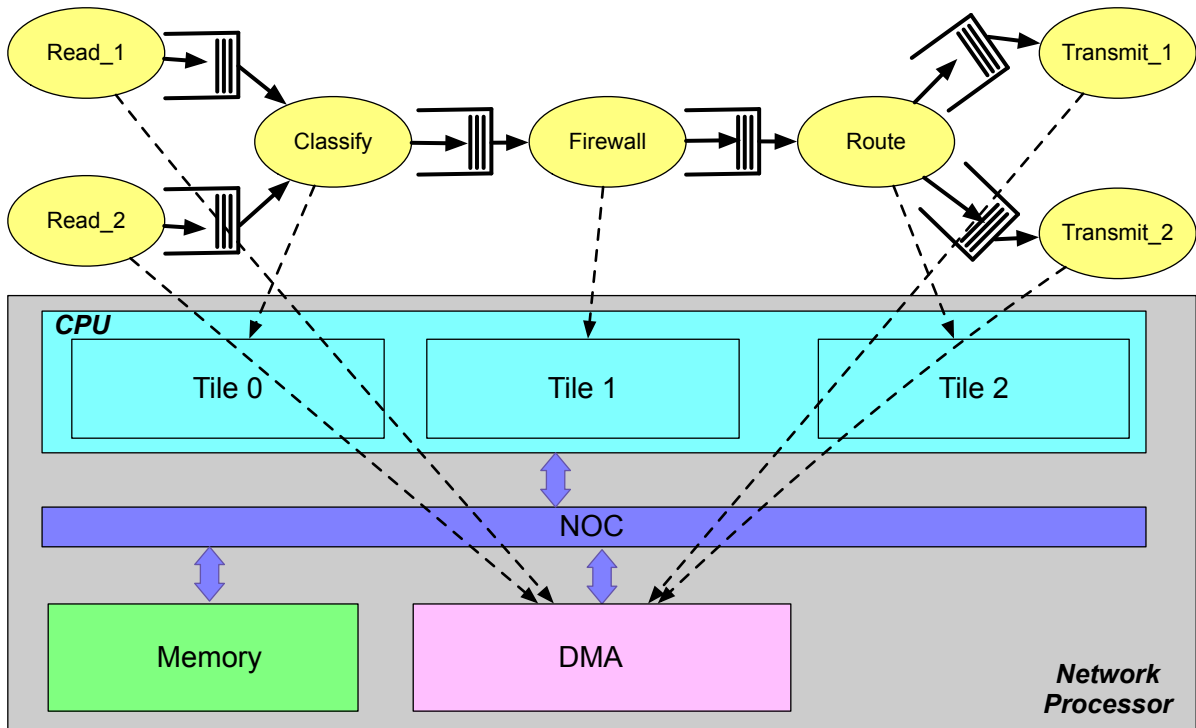


Figure 5.12: *Configuration\_1*: Use case modeled in the form of task graphs (above) mapped on to network processor performance model (below) with 3 tiles of CPU and a DMA.

and small packets. The larger packets' inter-arrival time is higher than smaller packets that affect the packet processing rate of the network configuration.

Power dissipation results for this configuration are shown in Figure 5.15b. It is interesting to see that the power consumption of the 256-byte packet is the lowest as soon as it reaches the max throughput at 500 MHz. Similarly, the power consumption for 128-byte packets is less than the 64-byte packet after achieving max throughput.

## 5.6 Comparison with Platform Architect

In order to evaluate the accuracy of the proposed methodology POEM, we compare it with state of the art Synopsys [16] tool called platform architect ultra (PA Ultra) [95]. PA Ultra was initially developed by CoWare and recently acquired by Synopsys to perform electronic system level (ESL) modeling of extra-functional properties like performance and power. PA Ultra and POEM methodologies are very similar. Therefore a comparative study can tell us about the equivalency of POEM.

There are few other EDA tools available, i.e., Intel Confluent [96] to evaluate a developing architecture for power and performance, one of the reasons behind opting PA Ultra is its

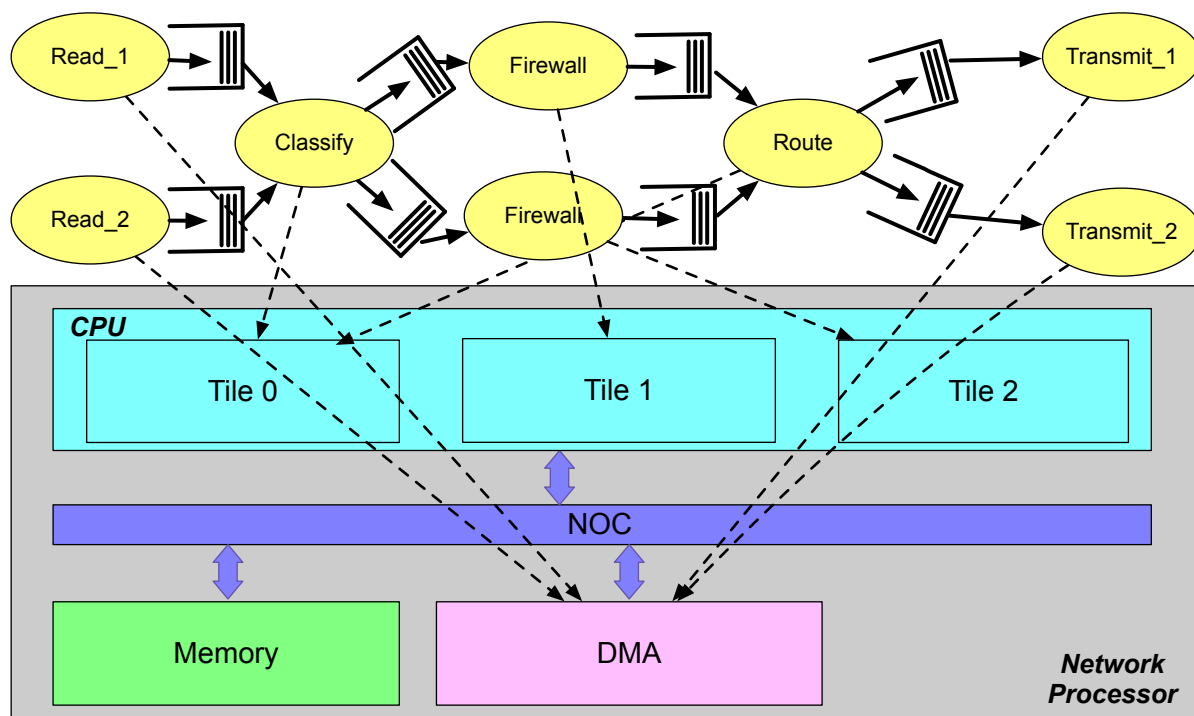


Figure 5.13: *Configuration\_2*: Use case modeled in the form of task graphs (above) mapped on to network processor performance model (below) with 3 tiles of CPU and a DMA. The difference from *Configuration\_1* is that the single firewall task is split into two nodes and mapped to two different tiles.

distributed setup just like POEM. It separates the power from application and performance. PA uses the IEEE standard format unified power format (UPF) [96], and the power database is attached using Tcl scripts.

Network processor example modeled in Section 5.5 is used for the comparison. The network processor example modeled in Section 5.5 is used for the comparison. First, the PA Ultra modeling flow is explained, and then how the use case is modeled, and at the end, a comparison of processing element usage and power consumption is made.

PA Ultra has SystemC model libraries for the most commonly used components called virtual processing units (VPU) (see Figure 5.16). Similarly, the use case (workload in PA Ultra) is modeled with task graphs (as shown in Figure 5.17) and mapped to VPUs. Task graphs and VPUs are configurable, and we have tried to model them as closely as possible to our application and performance models. PA uses UPF to annotate an electrical power design with the help of Tcl script for combined performance and power analysis.

The results demonstrate a close correlation between the loads of the components and their power consumption calculated by POEM for *Configuration\_1*, as shown in Table 5.8.

It is worth mentioning that POEM's simulation speed is six times faster than PA Ultra, as the simulation time took was three and eighteen seconds for network processor example,

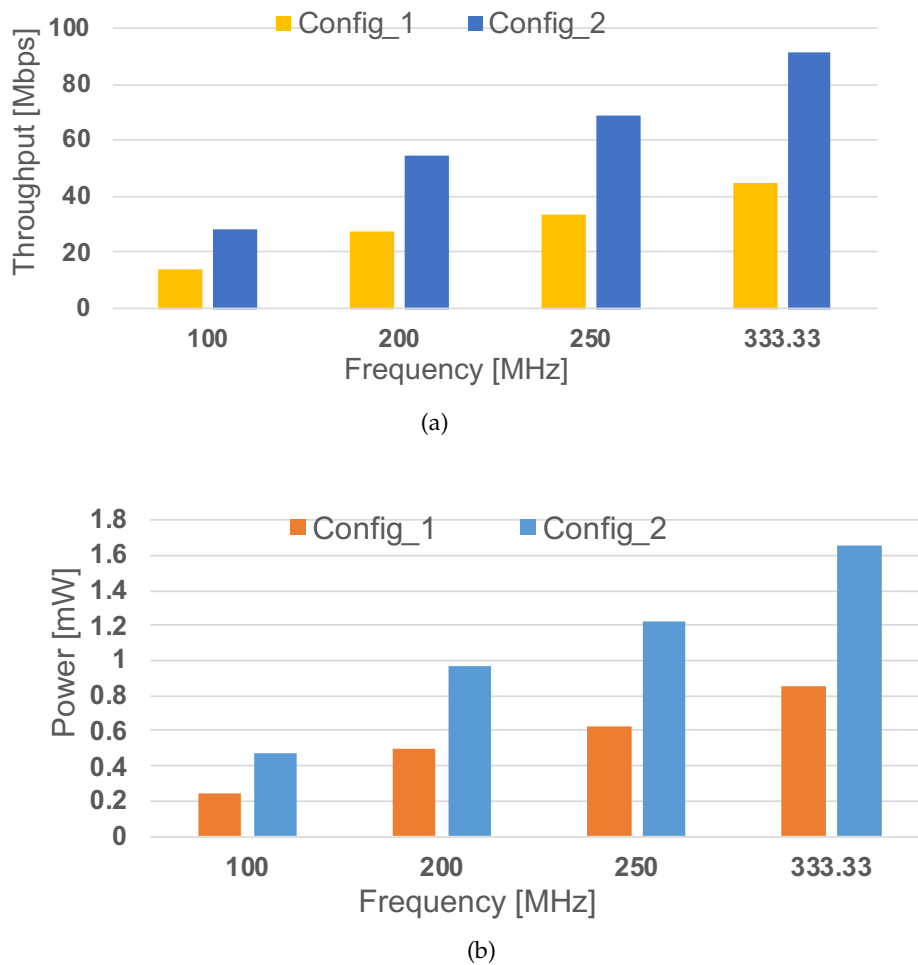
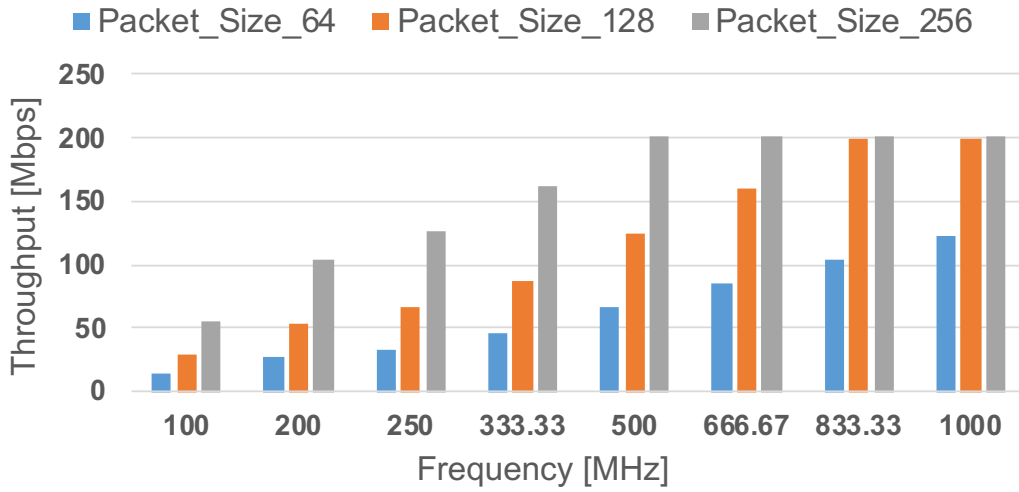


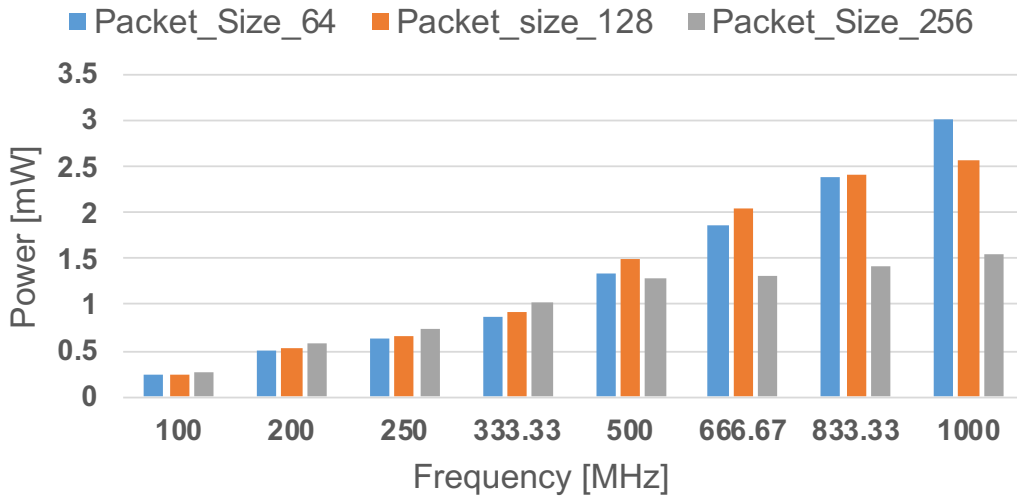
Figure 5.14: Throughput and power numbers, a) throughput of configurations over frequency for 64-byte packet size, b) power dissipation over frequency for 64-byte packet size.

respectively. There could be multiple reasons behind, i.e generic VPUs, obtaining the license for PA, reading the Tcl script to attach the power database etc.

The difference and advantage lie in the separation of concerns principle as POEM uses best in class performance and power modeling solutions. POEM maps task graph to specialized components, and PA Ultra maps task graph to generic VPUs. Hence, processing elements are more flexible, scale-able (for complex NoC architecture developed), and tailored to a use case in POEM than PA Ultra. Similarly, instead of working with UPF file, POEM develops a power model in a dedicated EDA tool with a web-based, collaborative power modeling and simulation framework that improves power roll-up productivity and early power architecture exploration.



(a)



(b)

Figure 5.15: Throughput and power numbers, a) Throughput for different packet sizes,  $V_{dd} = 0.9$  V,  $t = 4$  ms, b) power dissipation for different packet sizes,  $V_{dd} = 0.9$  V,  $t = 4$  ms.



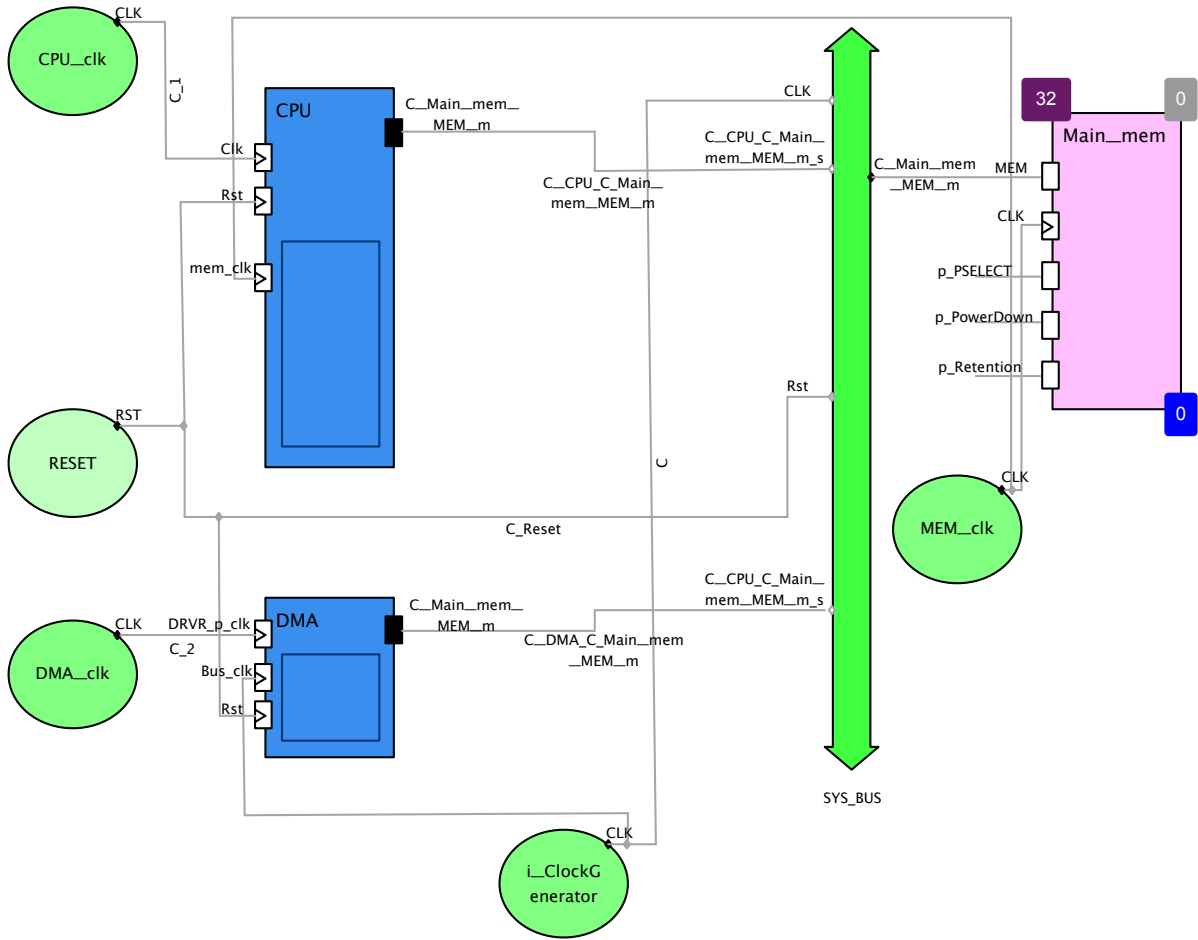


Figure 5.16: A top-level view of modeled network processor hardware in PA Ultra. CPU and DMA are made of generic VPUs and configured to perform specified functions.

Table 5.8: Comparison between POEM and PA Ultra

Component	POEM Power[W]	PA Ultra Power[W]	delta[%]
Tile 0	1.64	1.59	+3.14
Tile 1	1.18	1.21	-2.58
Tile 2	0.98	1.05	-6.67
Memory	2.39	2.29	+4.36
Component	POEM Usage[%]	PA Ultra Usage[%]	delta[%]
Tile 0	99.9	99.9	0
Tile 1	71.8	76	-4.8
Tile 2	59.4	65.9	-6.5
Memory	64.5	60.1	+7.15

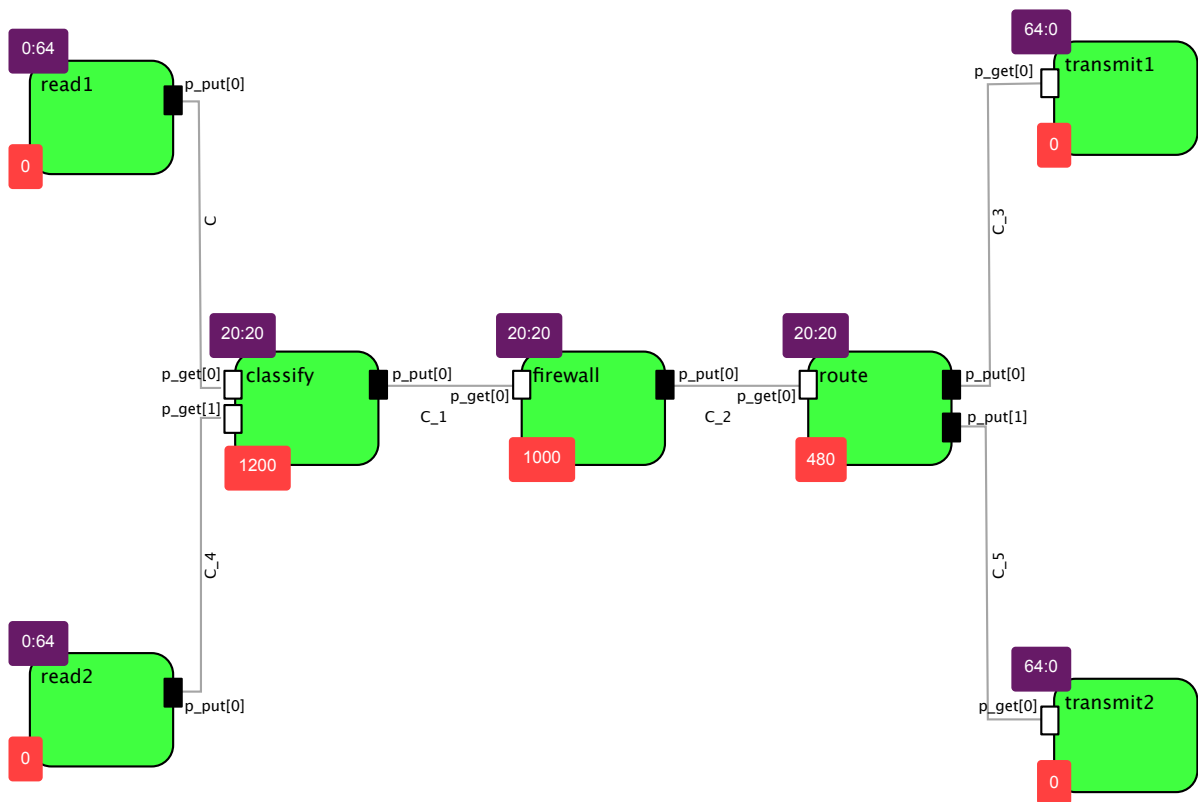


Figure 5.17: Modeling of network processor's application with task graph in PA Ultra. Seven tasks and their sequence are depicted, and it can be seen that they have identical order and sequence as in POEM.

## 6 Integration of the Methodology At Software and Hardware Abstraction Levels

Implementing the introduced methodology at software and hardware abstraction level is crucial to continue power and performance analysis and co-optimization throughout the embedded system design cycle. As power and performance optimization and exploration methodology (POEM) targets the holistic and multi-abstraction analysis and optimization, this chapter describes the aspects targeted and gaps filled with our methodology's implementation at these (software and hardware) abstraction levels. A detailed description and definition of the software and hardware abstraction levels have been introduced in Section 2.3.

As this research work collaborates with Intel, the implementation of the methodology (POEM) at the software and hardware abstraction level is influenced by their custom needs, challenges faced, and practices. Nevertheless, this chapter explains the generic implementation steps required for POEM to work at these abstraction levels without giving away Intel's trade secrets.

### 6.1 POEM at Software Abstraction Level

For fair comparison and to get the perspective of implementation efforts required at the software abstraction level (SAL), it is necessary to evaluate the methodology (POEM) based on the implementation of the same essential components of the methodology that are introduced at the architecture abstraction level (AAL) in Chapter 4. Hence, first, the application, then performance, and at the end, power model implementation details are discussed.

At software abstraction level, the embedded system's software is developed or matured enough to simulate system level use case or application or a scenario under analysis. Unlike at architecture abstraction level, the application is not modeled with task graphs, and the accuracy of the application model is high as well. The disadvantage is if any change is required in the application model at software abstraction level, higher efforts are required. Furthermore, several checks are in place as a standard practice in the industry to ensure that the new changes made in the application model (software) do not break any other functional component.

That implies the advantage of rapid prototyping is comprised at the cost of higher accuracy of the application model. So, in POEM at software abstraction level, the existing software written in embedded C is used and fulfills the application model's need.

The second essential component of the system level methodology after the application model is the simulation environment development by modeling the vital architecture components, known as the performance model. POEM uses virtual prototype as an architecture

model (developed using SystemC TLM libraries) at software abstraction level. Virtual prototype is very similar to the performance environment introduced in architecture abstraction level. Except most of virtual prototype 's components are modeled at loosely-timed (LT) transaction-level modeling, unlike approximately-timed in architecture abstraction level performance model. Virtual prototype is a prototype of a complete platform, and in order to rapidly verify the functionality, speed gain due to loosely-timed modeling is of utmost importance.

In POEM, virtual prototype 's crucial architecture components for a system level use case are modified with an ISCTLM (see details in section 3.1.2) wrapper (to enable run-time simulation configuration with attributes and tracing mechanism) to make the architecture model consistent for interface and mapping with the power modeling framework.

The power model is developed independently of the application and architecture models in Docea™ (3.3.1). Hence, its modeling and implementation will not be affected by the application model being replaced by the standard software at software abstraction level. It is straight forwards that software abstraction level has more low-level implementation details available; therefore, the power model developed in Docea for software abstraction level is fine-grained.

The Figure 6.1 shows the flow of the POEM at software abstraction level. There are five steps involved and explained as follow:

- Step 1: Virtual prototype of the embedded system is loaded with the production software binaries.
- Step 2: System level use case or a scenario is triggered, and with ISCTLM instrumentation, stimuli of relevant processing elements are collected in a value change dump file as an outcome of SystemC simulation. Performance key performance indicators like throughput, back pressure, latencies, and high-level activities can already be observed from this SystemC simulation.
- Step 3: Based on the value change dump information, a java script object notation (JSON) file is prepared. The java script object notation file contains the input and output definition of the power modeling framework and also serves as the mapping function of functional to power states.
- Step 4: Docea™ simulates the value change dump file for power analysis and produces a power trace as an output.
- Step 5: Analysis of power traces over time to identify the potential optimization opportunities.

## 6.2 POEM at Hardware Abstraction Level

Implementation of POEM at hardware abstraction level (HAL) is no different from software abstraction level and architecture abstraction level. Nevertheless, hardware abstraction level's

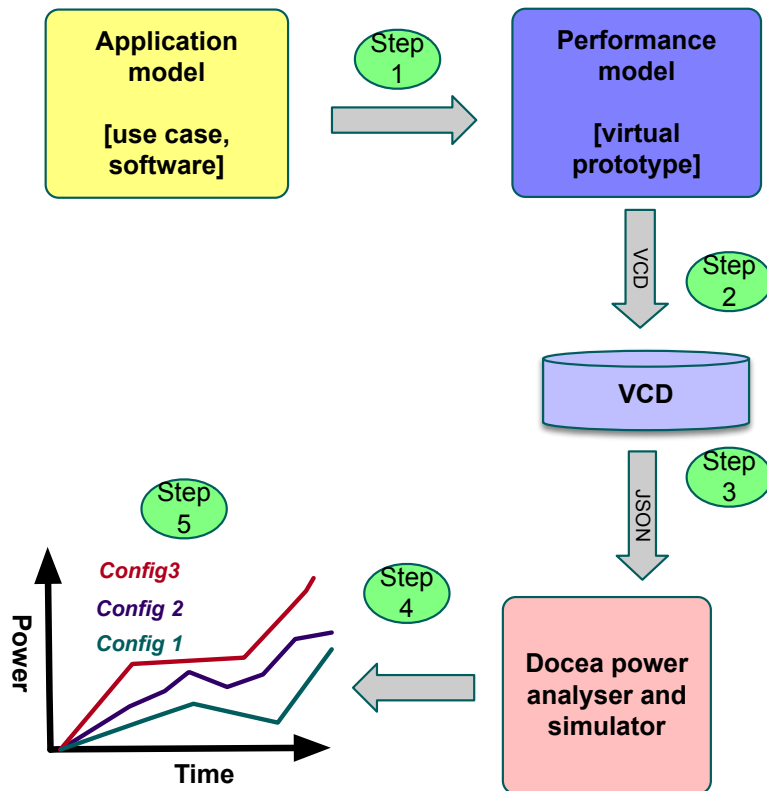


Figure 6.1: POEM: A block-level overview of the POEM at software abstraction level. This modular setup is capable of step-by-step refinements, whether application development, architecture resource selection, or average power analysis.

focus is to use Docea™ as a power simulator rather than the actual hardware. During the final stage of the embedded system design process, multiple hardware prototypes are developed before the actual tape-out to verify the production software and meet specific industry standards to get the platform's certification. These prototypes are not capable of measuring power consumption without special instrumentation.

Using Docea™ to measure the power for a system level use case at hardware abstraction level by running the standard software on a hardware prototype can give power consumption at a fine-grain level. Even if a particular hardware prototype is prepared to measure the power consumption of the critical system level power use case, it is limited by the voltage rails and the platform's power domain division constraints. Embedded system consists of many power and clock domains, components having the same power or clock domain share the same voltage and clock rails, respectively. The only available IEEE standard format for power modeling (unified power Format (UPF) [96]) uses the power domain concept to model the multiprocessors system on chips for power.

To understand the problem statement better, let us take the example of a cellular platform for a power critical system level use case; it is required to measure a component's power consumption in an isolated fashion. However, if the power domain in which the component

is located has other components, there is no way to calculate its power contribution. The differential approach often used to estimate the power consumption between two simulation runs is not applicable in this scenario; the chances are that the measurement fluctuations are higher than the power consumption difference between them.

Figure 6.2 shows the abstract overview of the POEM at hardware abstraction level to measure the power consumption using Docea™ following below mentioned steps.

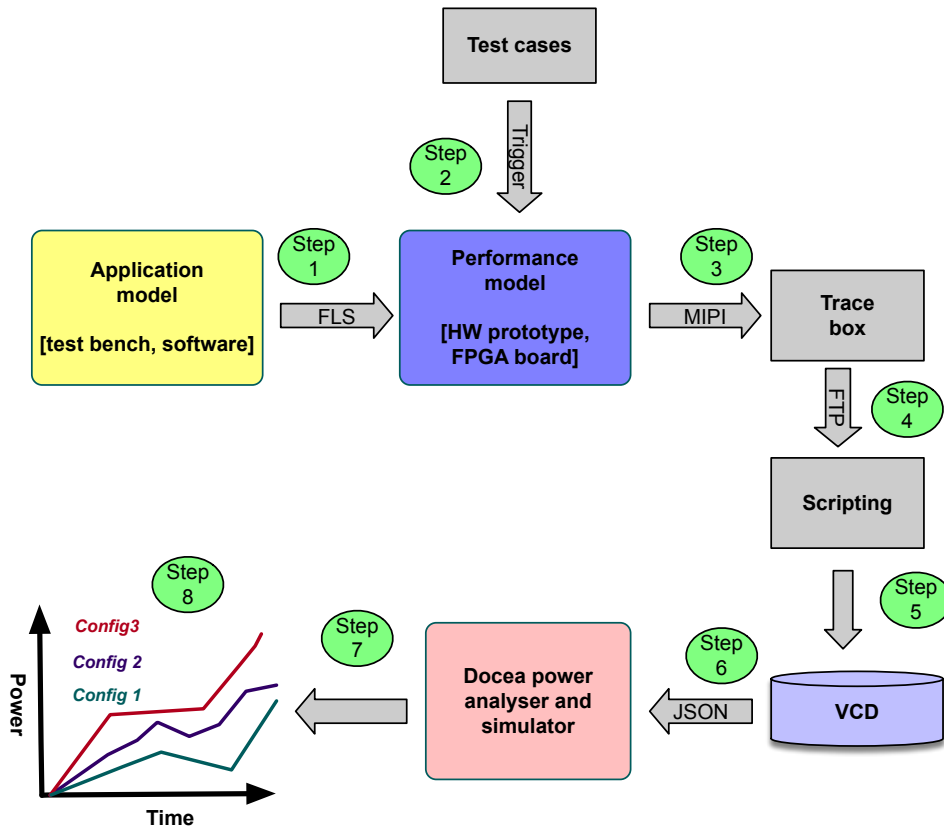


Figure 6.2: A block-level overview of the POEM at a hardware abstraction level: How are test-cases triggered? After flashing the software or use case on the performance model (that consists of FPGA or prototype hardware at a hardware abstraction level). Next, collection of the relevant parameters and post-processing with scripts to get the value change dump file use able by power molding framework to re-simulate and generate power trace over time.

Step 1: The production software is flashed on prototype hardware or field programmable gate array (FPGA) board.

Step 2: The performance model consisting of SystemC modules in architecture abstraction level and software abstraction level are replaced with actual hardware components, and in order to trigger the system use case, a host is required to send the commands to the evaluation board acting as a performance model.

- Step 3: System-level use cases are triggered, and traces are captured in a trace box.
- Step 4: Extract the traces and open them in a tool for functional verification of the system use case and debugging if required before passing it for post-processing.
- Step 5: System traces captured during the last step require post-processing to make them use-able for the power modeling framework as this approach keeps the methodology modular. The information captured in the form of traces is transformed into a value change dump file with scripts and later used in the Docea.
- Step 6: Based on the value change dump information, a java script object notation file is prepared. The java script object notation file contains the input and output definition of the power modeling framework and also serves as the mapping function of functional to power states.
- Step 7: Docea™ simulates the value change dump file for power analysis and produces a power trace as an output.
- Step 8: Analysis of power traces over time to identify the potential optimization opportunities.

# 7 Conclusion and Future Work Directions

## 7.1 Conclusion

POEM is a methodology for power and performance estimation for complex systems, like, for instance, mobile platforms. The key advantage over state of the art is the explicit mapping methodology between performance models and the power models, along with the modeling of system level use case with task graphs in the first place.

This dissertation's outcome backs the motivation and problem statement; power and performance estimation, exploration, and co-optimization for the sophisticated and state of the art embedded system along with coupling of different design abstraction levels for key performance indicator gains.

Division and definition of embedded system design cycle into three distinct abstraction levels are influenced by current industry practices' needs and gaps. Nevertheless, the holistic methodology spanning the complete design cycle is the preferred solution irrespective of the abstraction level definitions introduced in this work.

Similarly, the idea to keep the methodology modular and exercises the principle of separation-of-concern principle is vital. It gives system engineers the freedom to nail down their module's configuration and design without being dependent on others' outcomes. Therefore, the modular setup of POEM enables model co-development and team collaboration and maximizes the reuse of existing models for a power performance integrated analysis.

Performance and power co-optimization has an added challenge in terms of finding out the right system level use case. It is challenging to identify the system level use cases critical for both power and performance points of view. This challenge is because performance critical use case targets maximum throughput, and critical power use cases focus on low throughput and idle periods.

## 7.2 Future Work Directions

The holistic, multi-abstraction and modular approach of POEM prefers offline connection between functional stimuli and power modeling framework. The integration of state-residencies is currently based on an offline connection (based on functional traces), which can be shared asynchronously. This enables separate refinements of the power model, e.g. for technology scaling or power domain definition.

An integrated simulation will also support feedback loops, where the use case may demand it. The Docea has started providing support for APIs, and these APIs can establish a connection with the SystemC module of application and performance models. Future work



may include automating the design space exploration against certain power and performance constraints set by the designers.

As Docea also provides a thermal solver, one future direction could be to integrate the thermal analysis with power and performance. Transition power modeling is not part of POEM. In the future, this missing contributor can be included in power modeling.

# List of Figures

2.1	Y-chart shown as a concentric circles, representing different abstraction level in three different domains [7]. The innermost circle corresponds to the lowest abstraction level, and the abstraction level increases as the distance of the concentric circles from the center increases. Fast and accurate system modeling of heterogeneous, multicore system on chip needs to be done at the highest abstraction level: system level. It corresponds to actual chip or prototype boards in the physical domain, systems in the behavioral domain (even above algorithms), and CPU, memory blocks in the structural domain. . . . .	7
2.2	Contemporary design flow of an embedded system and an overview of the relation between design abstraction, design detail and simulation time: The embedded system is divided into four abstraction levels. Simultaneously, design specifications sit at the top, and the system level, also known as the electronics system level (ESL), is the first abstraction level derived based on customer-provided design specifications. The abstraction level is the highest, the time required for a system level use case is the lowest, and very few design details are known at this point. The hierarchy of the funnels is to visualize the available field of design space exploration. The complete design space exploration field is accessible from the top where the design specification resides. These streams are further divided into three abstraction levels. Register transfer level Design is the first, and the transistor level is the last abstraction level in the hardware stream. Similarly, software written in C/C++ is the first, and executable is the last abstraction level in the software stream. . . . .	8
2.3	POEM: A multi-abstraction level methodology, estimating KPIs at architectural, software, and hardware abstraction level with a common power modeling framework. That implies a methodology in which the modules addressing functional simulation, power modeling, or application development are agnostic concerning each other. . . . .	28
3.1	Node of a task graph: Input is the abstract communication data coming from SC_fifos connected to the input ports and finally ending up in input buffers. The method/function SC_THREAD controls the node behavior and processes the data to the output buffer. The output buffers transfer data to the output FIFO channels. According to the provided mapping information, the Req_start member of the task graph node requests execution of the task on an underlying architecture. . . . .	34

3.2	Illustration of source, sink, internal and autonomous nodes. FIFO buffers (communication channels) between these show the relation between input and output ports. . . . .	35
3.3	A piece of configuration file depicting a task's specification using all or a subset of task graph node parameters. . . . .	37
3.4	During the first step, the node waits for the release time of the task to expire, and in the second step, it waits for input ports to read the communication data. The third step is the core execution state of the execution flow, and the task graph node can take <i>ACTIVE</i> , <i>PREEMPTED</i> , or <i>PENDING</i> states depending upon the situation. Fourth is the final step, and the task graph node finishes its job and puts execution flow into the <i>COMPLETED</i> state. . . . .	39
3.5	A configuration file is shown (right), consisting of three entries for task priority, period, and debug option. And the corresponding hierarchy of the class in the configuration file is shown on the left side. . . . .	41
3.6	Task graph creation flow: It consists of four steps and starts with the registration of all the resources to instantiating task graph nodes to configuration nodes' internal behavior to map the task to architecture model to connection establishment. . . . .	42
3.7	A piece of configuration file depicting three tasks and their specifications using all or a subset of task graph node parameters. . . . .	44
3.8	An enhanced (internal) view of the Task Scheduler Interface. It acts as a Real-time operating system as it controls the functionality and manages the data-flow between the task graph nodes and processing elements. . . . .	47
3.9	Virtual Engine processing element internal flow: In a Virtual Engine, the execution flow starts with a wait state. The start signal notifies about the task to be simulated. It could be a new task or an already running task that resumes. The wait statement in SystemC simulates the execution time of the current task in a Virtual Engine. The wait state of a task graph node is preemptable by a SystemC event, given that the execution time is greater than the current waited time at the point of the SystemC event preempting the task. As soon as the waited time is greater than the execution time, it ends. . . . .	49
3.10	Docea flow: At the top of the hierarchy sits System, Subsystem, or Component. A System contains the Subsystems or Components to give the flexibility and ease of modeling components and reusability in different projects. . . . .	51
3.11	Generation of computable power model from a web-based Docea™ Power Analytics: An offline script produces a computational model using power interface information in the form of a JSON file. It is also seen as a Python power calculator. It gives in-feed to Docea™ Power Simulator and Python environment configurable by control power input parameters, i.e., voltage, frequency, modes, activity ratio. [89] . . . . .	53

4.1	POEM: Layer-1) application modeled as a task graph mapped to processing elements of the performance model. Layer-2) SystemC simulation of Layer-1 generates performance and power stimuli, mapped to the power database, and re-simulated for power vs. time for different configurations. . . . .	56
4.2	Abstract representation of main functional blocks of a cellular modem (below) and of a task graph mapped to the blocks (above). . . . .	59
4.3	Mapping of the hardware accelerator’s stimuli (in the form of VCD) generated as a result of SystemC simulation of the performance model, on to the Coprocessor in the power model with the help of JSON file. . . . .	64
5.1	Network application task graph sequence shown with seven task graph nodes and FIFO channels in between them after analyzing with a virtual engine. . .	73
5.2	System’s corresponding data throughput (a) and packet throughput (b) (consisting of a single virtual engine) at different input data rates and packet sizes. . . . .	75
5.3	Virtual engine usage by each task for maximum throughput. Firewall Classify and Route are the three tasks that contribute towards the virtual engine usage, respectively, for all three packet sizes. . . . .	76
5.4	System’s corresponding data throughput (a) and packet throughput (b) (consisting of multiple virtual engines) at different input data rates and packet sizes. . . . .	77
5.5	Average execution latency of the tasks on the corresponding engines for 500 MHz (a) and 1.5 GHz (b). . . . .	78
5.6	Average execution latency of the tasks on the corresponding engines for 2 CPU core (1.5 GHz) configuration. . . . .	80
5.7	System’s performance for 2 CPU cores (1.5 GHz) a) data throughput, b) packet throughput. . . . .	81
5.8	System’s performance for 3 CPU cores at 1.5 GHz frequency at three different packet sizes, the system run with 256-bytes packet size can archive max output throughput. . . . .	83
5.9	Average execution latency of the tasks on their corresponding engines for 3 CPU core (1.5 GHz) configuration. . . . .	83
5.10	Average execution latency of the tasks on their corresponding engines for 3 CPU core (1.5 GHz) with improved memory block. . . . .	84
5.11	System performance for 3 CPU cores (1.5 GHz) with improved memory block.	85
5.12	<i>Configuration_1</i> : Use case modeled in the form of task graphs (above) mapped on to network processor performance model (below) with 3 tiles of CPU and a DMA. . . . .	87
5.13	<i>Configuration_2</i> : Use case modeled in the form of task graphs (above) mapped on to network processor performance model (below) with 3 tiles of CPU and a DMA. The difference from <i>Configuration_1</i> is that the single firewall task is split into two nodes and mapped to two different tiles. . . . .	88

5.14	Throughput and power numbers, a) throughput of configurations over frequency for 64-byte packet size, b) power dissipation over frequency for 64-byte packet size. . . . .	89
5.15	Throughput and power numbers, a) Throughput for different packet sizes, $V_{dd} = 0.9\text{ V}$ , $t = 4\text{ ms}$ , b) power dissipation for different packet sizes, $V_{dd} = 0.9\text{ V}$ , $t = 4\text{ ms}$ . . . . .	90
5.16	A top-level view of modeled network processor hardware in PA Ultra. CPU and DMA are made of generic VPUs and configured to perform specified functions. . . . .	91
5.17	Modeling of network processor's application with task graph in PA Ultra. Seven tasks and their sequence are depicted, and it can be seen that they have identical order and sequence as in POEM. . . . .	92
6.1	POEM: A block-level overview of the POEM at software abstraction level. This modular setup is capable of step-by-step refinements, whether application development, architecture resource selection, or average power analysis. . . . .	95
6.2	A block-level overview of the POEM at a hardware abstraction level: How are test-cases triggered? After flashing the software or use case on the performance model (that consists of FPGA or prototype hardware at a hardware abstraction level). Next, collection of the relevant parameters and post-processing with scripts to get the value change dump file use able by power molding framework to re-simulate and generate power trace over time. . . . .	96

## List of Tables

3.1	A task execution workload ( <code>workload_obj</code> ) mapped on a Virtual Engine . . .	31
3.2	A typical task execution workload ( <code>workload_obj</code> ) mapped on a DMA channel	32
3.3	A typical task execution workload ( <code>workload_obj</code> ) mapped on a CPU with instruction and data caches . . . . .	33
3.4	General parameters of a task graph node . . . . .	36
3.5	Main function members of a task graph node . . . . .	37
3.6	Main function and variable members of <code>tg_constructor</code> . . . . .	43
4.1	Distinct functional states of a tile and corresponding power states . . . . .	63
5.1	Number of instructions of a network processor application tasks with CPI=1.2	71
5.2	Network processor specifications . . . . .	72
5.3	Virtual engine workload load for different tasks . . . . .	74
5.4	Use case modeled in the form of task graphs mapped on to single virtual engine	74
5.5	Use case modeled in the form of task graphs mapped on to 3 virtual engines .	76
5.6	Use case modeled in the form of task graphs mapped on to network processor performance model with 2 CPU cores and a DMA . . . . .	79
5.7	Use case modeled in the form of task graphs mapped on to network processor performance model with 3 CPU cores and a DMA . . . . .	82
5.8	Comparison between POEM and PA Ultra . . . . .	91

# Listings

- 3.1 Definition of input with value conversion, It is possible to connect an input to power model parameters and convert input values for each power model parameter. In the expression, the input value is identified by the tag input. . . 53
- 4.1 A piece of VCD file generated as a result of SystemC simulation of the performance model of the hardware accelerator. . . . . 64
- 4.2 JSON file for mapping of the hardware accelerator’s stimuli generated as a result of SystemC simulation of the performance model, on to the Co-processor. 67

## Bibliography

- [1] B. Raaf, M. Faerber, B. Badic, and V. Frascolla. "KEY TECHNOLOGY ADVANCEMENTS DRIVING MOBILE COMMUNICATIONS FROM GENERATION TO GENERATION." In: *Intel Technology Journal* 18.1 (2014).
- [2] G. E. Moore. "Cramming more components onto integrated circuits." In: *IEEE solid-state circuits society newsletter* 20.3 (2006).
- [3] J. Yang, X. Ge, and Y. Zhong. "How Much of Wireless Rates Can Smartphones Support in 5G Networks?" In: *IEEE Network* 33.3 (2018), pp. 122–129.
- [4] GSMA™ Intelligence. <https://www.gsmainelligence.com>. Accessed: 2020-02-23.
- [5] V. Ilderem. "1.4 5G Wireless Communication: An Inflection Point". In: *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2019, pp. 35–39.
- [6] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. "System-level design: orthogonalization of concerns and platform-based design". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 19.12 (2000), pp. 1523–1543.
- [7] D. Gajski and R. H. Kuhn. "New VLSI Tools - Guest Editors' Introduction". In: *IEEE Computer* 16.12 (1983), pp. 11–14.
- [8] Y. Veller and S. Matalon. "Why you should optimize power at the ESL-Whitepaper". In: {Online} <http://go.mentor.com/cvtq> (2010).
- [9] G. Martin, B. Bailey, and A. Piziali. *ESL design and verification: a prescription for electronic system level methodology*. Elsevier, 2010.
- [10] F. Ghenassia et al. *Transaction-level modeling with SystemC*. Springer, 2005.
- [11] P. R. Panda. "SystemC: a modeling platform supporting multiple design abstractions". In: *Proceedings of the 14th international symposium on Systems synthesis*. 2001, pp. 75–80.
- [12] L. Cai, D. Gajski, and M. Olivarez. "Introduction of system level architecture exploration using the SpecC methodology". In: *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196)*. Vol. 5. IEEE. 2001, pp. 9–12.
- [13] C. Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.
- [14] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.
- [15] R. K. Gupta. *Co-synthesis of hardware and software for digital embedded systems*. Vol. 329. Springer Science & Business Media, 2012.



- [16] Synopsys®. <https://www.synopsys.com>. Accessed: 2019-02-23.
- [17] Cadence® design systems. <https://www.cadence.com>. Accessed: 2019-02-23.
- [18] A. Donlin. "Transaction level modeling: flows and use models". In: *International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES+ ISSS 2004*. IEEE. 2004, pp. 75–80.
- [19] T. Kogel. "Tlm peripheral modeling for platform-driven esl design". In: *Technical report, CoWare Inc.* (2006).
- [20] J. Aynsley et al. "OSCI TLM-2.0 language reference manual". In: *Open SystemC Initiative (OSCI)* (2009), p. 15.
- [21] Y. Yi, D. Kim, and S. Ha. "Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization". In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 26.12 (2007), pp. 2186–2200.
- [22] A. Chandrakashan and R. Brodersen. *Low power digital CMOS design*. 1996.
- [23] M. Sharma, R. Gautam, and M. A. Khan. *Design and Modeling of Low Power VLSI Systems*. IGI Global, 2016.
- [24] S. McCloud. "Low-Power RTL Report". In: *Calypto Design Systems* (2012).
- [25] L. Benini and G. De Micheli. "Automatic synthesis of low-power gated-clock finite-state machines". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.6 (1996), pp. 630–643.
- [26] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou. "Precomputation-based sequential logic optimization for low power". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2.4 (1994), pp. 426–436.
- [27] L. W. Nagel. "SPICE2: A computer program to simulate semiconductor circuits". In: *Ph. D. dissertation, University of California at Berkeley* (1975).
- [28] P. E. Design. "Tools Group". In: *Philips Research, DIESEL User Manual, version 2* (2001).
- [29] R. P. Llopis and K. Goossens. "The petrol approach to high-level power estimation". In: *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*. IEEE. 1998, pp. 130–132.
- [30] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: a first step towards software power minimization". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2.4 (1994), pp. 437–445.
- [31] J. Laurent, N. Julien, E. Senn, and E. Martin. "Functional level power analysis: An efficient approach for modeling the power consumption of complex processors". In: *Proceedings of the conference on Design, automation and test in Europe-Volume 1*. IEEE. 2004, p. 10666.
- [32] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas. "Instruction level energy modeling for pipelined processors". In: *Journal of Embedded Computing* 1.3 (2005), pp. 317–324.

- [33] E. Senn, J. Laurent, N. Julien, and E. Martin. "SoftExplorer: estimation, characterization, and optimization of the power and energy consumption at the algorithmic level". In: *Intl. workshop on power and timing modeling, optimization and simulation*. Springer. 2004, pp. 342–351.
- [34] V. S. Adve and M. K. Vernon. "Parallel program performance prediction using deterministic task graph analysis". In: *ACM Transactions on Computer Systems (TOCS)* 22.1 (2004), pp. 94–136.
- [35] S. Chakraborty, S. Künzli, and L. Thiele. "A General Framework for Analysing System Properties in Platform-Based Embedded System Designs." In: *Date*. Vol. 3. Citeseer. 2003, p. 10190.
- [36] M. Gries, C. Kulkarni, C. Sauer, and K. Keutzer. "Exploring trade-offs in performance and programmability of processing element topologies for network processors". In: *Network Processor Design: Issues and Practices 2* (2003), pp. 1–5.
- [37] L. Thiele, S. Chakraborty, M. Gries, and S. Kunzli. "4 Design Space Exploration of Network Processor". In: *Network Processor Design: Issues and Practices 1* (2003).
- [38] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. "Embedded software in network processors—models and algorithms". In: *International Workshop on Embedded Software*. Springer. 2001, pp. 416–434.
- [39] L. Benini, R. Hodgson, and P. Siegel. "System-level power estimation and optimization". In: *Proceedings of the 1998 international symposium on Low power electronics and design*. 1998, pp. 173–178.
- [40] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich. "ESL power and performance estimation for heterogeneous MPSoCs using SystemC". In: *FDL 2011 Proceedings*. IEEE. 2011, pp. 1–8.
- [41] S. Honnavara-Prasad. "Overview of IEEE1801-2015: Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. 2016, pp. 186–186.
- [42] A. A. Garcia, J. Gobert, T. Dombek, H. Mehrez, and F. Petrot. "Cycle-accurate energy estimation in system level descriptions of embedded systems". In: *9th international conference on electronics, circuits and systems*. Vol. 2. IEEE. 2002, pp. 549–552.
- [43] T. Diop, N. E. Jerger, and J. Anderson. "Power modeling for heterogeneous processors". In: *Proceedings of workshop on general purpose processing using GPUs*. 2014, pp. 90–98.
- [44] L. Benini, A. Bogliolo, M. Favalli, and G. De Micheli. "Regression models for behavioral power estimation". In: *Integrated Computer-Aided Engineering* 5.2 (1998), pp. 95–106.
- [45] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. "An accurate instruction-level energy consumption model for embedded risc processors". In: *Acm Sigplan Notices* 36.8 (2001), pp. 1–10.

- [46] S. K. Rethinagiri, R. B. Atitallah, and J.-L. Dekeyser. "A system level power consumption estimation for mp soc". In: *2011 International Symposium on System on Chip (SoC)*. IEEE. 2011, pp. 56–61.
- [47] S. E. Lee and N. Bagherzadeh. "A high level power model for Network-on-Chip (NoC) router". In: *Computers & Electrical Engineering* 35.6 (2009), pp. 837–845.
- [48] P. Lieverse, P. Van Der Wolf, K. Vissers, and E. Deprettere. "A methodology for architecture exploration of heterogeneous signal processing systems". In: *Journal of VLSI signal processing systems for signal, image and video technology* 29.3 (2001), pp. 197–207.
- [49] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. "System level design with SPADE: an M-JPEG case study". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*. IEEE. 2001, pp. 31–38.
- [50] V. D. Zivkovic, E. Deprettere, P. Van der Wolf, and E. De Kock. "Design space exploration of streaming multiprocessor architectures". In: *IEEE Workshop on Signal Processing Systems*. IEEE. 2002, pp. 228–234.
- [51] V. D. Zivkovic, E. Deprettere, E. De Kock, and P. van der Wolf. "Fast and accurate multiprocessor architecture exploration with symbolic programs". In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE. 2003, pp. 656–661.
- [52] E. A. de Kock, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzter, P. Lieverse, K. A. Vissers, and G. Essink. "YAPI: Application modeling for signal processing systems". In: *Proceedings of the 37th Annual Design Automation Conference*. 2000, pp. 402–405.
- [53] A. Turjan, B. Kienhuis, and E. Deprettere. "Translating affine nested-loop programs to process networks". In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. 2004, pp. 220–229.
- [54] A. D. Pimentel, C. Erbas, and S. Polstra. "A systematic approach to exploring embedded system architectures at multiple abstraction levels". In: *IEEE Transactions on Computers* 55.2 (2006), pp. 99–112.
- [55] H. L. Muller et al. *Simulating computer architectures*. Citeseer, 1993.
- [56] T. Wild, A. Herkersdorf, and G.-Y. Lee. "TAPES—Trace-based architecture performance evaluation with SystemC". In: *Design Automation for Embedded Systems* 10.2-3 (2005), pp. 157–179.
- [57] A. Bakshi, V. K. Prasanna, and A. Ledeczi. "MILAN: A model based integrated simulation framework for design of embedded systems". In: *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*. 2001, pp. 82–93.
- [58] S. Mohanty and V. K. Prasanna. "Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures". In: *15th Annual IEEE International ASIC/SOC Conference*. IEEE. 2002, pp. 160–167.

- [59] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. "Metropolis: An integrated electronic system design environment". In: *Computer* 36.4 (2003), pp. 45–52.
- [60] C. Jaber, A. Kanstein, L. Apvrille, A. Baghdadi, P. Le Moenner, and R. Pacalet. "High-level system modeling for rapid hw/sw architecture exploration". In: *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*. IEEE. 2009, pp. 88–94.
- [61] L. Apvrille, W. Muhammad, R. Ameer-Boulifa, S. Coudert, and R. Pacalet. "A UML-based environment for system design space exploration". In: *2006 13th IEEE International Conference on Electronics, Circuits and Systems*. IEEE. 2006, pp. 1272–1275.
- [62] B. Kienhuis, F. Deprettere, P. van der Wolf, and K. Vissers. "The Y-chart approach". In: *Embedded processor design challenges*. Springer. 2002, p. 18.
- [63] M. Waseem, L. Apvrille, R. Ameer-Boulifa, S. Coudert, and R. Pacalet. "Abstract application modeling for system design space exploration". In: *9th EUROMICRO Conference on Digital System Design (DSD'06)*. IEEE. 2006, pp. 331–337.
- [64] S. Mahadevan, K. Virk, and J. Madsen. "ARTS: A SystemC-based framework for multi-processor systems-on-chip modelling". In: *Design Automation for Embedded Systems* 11.4 (2007), pp. 285–311.
- [65] A. Sarma. "Introduction to SDL-92". In: *Computer Networks and ISDN Systems* 28.12 (1996), pp. 1603–1615.
- [66] A. Baghdadi, N.-E. Zergainoh, W. O. Cesario, and A. A. Jerraya. "Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems". In: *IEEE Transactions on Software Engineering* 28.9 (2002), pp. 822–831.
- [67] A. Varma, E. Debes, I. Kozintsev, P. Klein, and B. Jacob. "Accurate and fast system-level power modeling: An XScale-based case study". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–20.
- [68] G. B. Vece and M. Conti. "Power estimation in embedded systems within a SystemC-based design context: the PKtool environment". In: *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*. IEEE. 2009, pp. 179–184.
- [69] D. Greaves and M. Yasin. "TLM POWER3: Power estimation methodology for SystemC TLM 2.0". In: *Models, Methods, and Tools for Complex Chip Design*. Springer, 2014, pp. 53–68.
- [70] T. Bouhadiba, M. Moy, and F. Maraninchi. "System-level modeling of energy in TLM for early validation of power and thermal management". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1609–1614.
- [71] T. Bouhadiba, M. Moy, F. Maraninchi, J. Cornet, L. Maillet-Contoz, and I. Materic. "Co-simulation of functional SystemC TLM models with power/thermal solvers". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 2176–2181.

- [72] M. Moy, C. Helmstetter, T. Bouhadiba, and F. Maraninchi. "Modeling Power Consumption and Temperature in TLM Models". In: *Leibniz Transactions on Embedded Systems* 3.1 (2016), pp. 03–1.
- [73] C. Helmstetter and M. Moy. "LIBTLMPWT: Model power-consumption and temperature in systemc/tlm". In: *Distributed under the terms of the GNU General Public License version 2* (2013).
- [74] R. Piscitelli and A. D. Pimentel. "A signature-based power model for mp soc on fpga". In: *VLSI Design 2012* (2012).
- [75] K. Gilles. "The semantics of a simple language for parallel programming". In: *Information processing* 74 (1974), pp. 471–475.
- [76] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, et al. "COMPLEX: Codesign and power management in platform-based design space exploration". In: *2012 15th Euromicro Conference on Digital System Design*. IEEE. 2012, pp. 349–358.
- [77] K. Grüttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel. "An ESL timing & power estimation and simulation framework for heterogeneous SoCs". In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. IEEE. 2014, pp. 181–190.
- [78] K. Grüttner, K. Hylla, S. Rosinger, and W. Nebel. "Towards an ESL framework for timing and power aware rapid prototyping of HW/SW systems". In: *2010 Forum on Specification & Design Languages (FDL 2010)*. IET. 2010, pp. 1–6.
- [79] C. Trabelsi, R. Ben Atitallah, S. Meftali, J.-L. Dekeyser, and A. Jemai. "A model-driven approach for hybrid power estimation in embedded systems design". In: *EURASIP Journal on Embedded Systems* 2011 (2011), pp. 1–15.
- [80] R. B. Atitallah, S. Niar, and J.-L. Dekeyser. "MPSoC power estimation framework at transaction level modeling". In: *2007 International Conference on Microelectronics*. IEEE. 2007, pp. 245–248.
- [81] A. Barreteau. "System-Level Modeling and Simulation with Intel® CoFluent™ Studio". In: *Complex Systems Design & Management*. Springer, 2016, pp. 305–306.
- [82] R. Dunford, Q. Su, and E. Tamang. "The pareto principle". In: *University of Plymouth* (2014).
- [83] A. S. Initiative et al. "IEEE 1666 Standard: SystemC Language Reference Manual., 2011". In: URL: <http://www.accellera.org> (2011).
- [84] H. Ahmadzay. "Design and Implementation of a Flexible Task Graph for Power and Performance Simulation Models". MA thesis. Technische Universität München, 2018.
- [85] M. M. Ayub, H. Ahmadzay, J. Eckmüller, and F. Kreupl. "Electronic System Level Power and Performance Analysis for Multi-Processor-System-on-Chip". In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2019, pp. 1–2.

- [86] R. Joseph, D. Brooks, and M. Martonosi. "Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs". In: *Workshop on Complexity Effective Design WCED, held in conjunction with ISCA*. Vol. 28. 2001.
- [87] M. Pedram and J. M. Rabaey. *Power aware design methodologies*. Springer Science & Business Media, 2002.
- [88] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin, and A. Sivasubramaniam. "vEC: virtual energy counters". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2001, pp. 28–31.
- [89] Intel® Docea™. <https://www.intel.com/content/www/us/en/system-modeling-and-simulation/docea/overview.html>. Accessed: 2019-02-23.
- [90] R. A. Bergamaschi and Y. W. Jiang. "State-based power analysis for systems-on-chip". In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE. 2003, pp. 638–641.
- [91] B. Kienhuis, E. F. Deprettere, P. Van der Wolf, and K. Vissers. "A methodology to design programmable embedded systems". In: *International Workshop on Embedded Computer Systems*. Springer. 2001, pp. 18–37.
- [92] A. B. Ameer, D. Martinot, P. Guitton-Ouhamou, V. Frascolla, F. Verdier, and M. Auguin. "Power and performance aware electronic system level design". In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2017, pp. 1–4.
- [93] K. Grüttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel. "An ESL timing & power estimation and simulation framework for heterogeneous SoCs". In: *(SAMOS XIV)*. July 2014, pp. 181–190. DOI: 10.1109/SAMOS.2014.6893210.
- [94] R. Ohlendorf. "A network processor architecture with application-optimized reconfigurable processing paths (FlexPath NP)". PhD thesis. Technische Universität München, 2011.
- [95] *Platform Architect by Synopsys®*. <https://www.synopsys.com/-verification/virtual-prototyping/platform-architect.html>.
- [96] P. Stanley-Marbell. "What is IEEE P1801 (unified power format)?" In: *ACM SIGDA Newsletter 37.19* (2007), pp. 1–1.

# Acronyms

- 3GPP** 3rd generation partnership project. 2
- AAL** architectural abstraction level. 55
- AP** application processor. 2
- API** application programming interface. 25
- AT** approximately-timed. 10, 26
- BCET** best-case execution time. 20
- CA** cycle-accurate. 9
- CDFG** control data flow graphs. 25
- CPU** central processing unit. 7, 100
- DAG** directed acyclic graphs. 25
- DMA** direct memory access. 79
- DSE** design space exploration. iii
- DVFS** dynamic voltage and frequency scaling. 5
- EDA** electronic design automation. 9, 26
- ESL** electronic system level. iii
- FLPA** functional level power analysis. 13
- FSM** finite state machine. 14
- GSMA** global system for mobile communications. 2
- GTL** generic task library. 23
- HAL** hardware abstraction level. 94
- HW** hardware. 3

**IDPA** Intel® Docea™ power analytics. 50

**IDPS** Intel® Docea™ power simulator. 50

**ILPA** instruction level power analysis. 13

**IMU** instruction management unit. 13

**JSON** java script object notation. 25

**KPN** kahn process networks. 25

**LT** loosely-timed. 10, 26

**MPSoCs** multiprocessors-system-on-chips. iii

**NNLS** non-negative least square. 15

**NoC** network on chips. 15

**OS** operating system. 10

**OSCI** open SystemC initiative. 10

**PA Ultra** platform architect ultra. 87

**POEM** power optimization and exploration methodology. 4, 27

**PSM** power state machine. 14

**PU** processing unit. 13

**PV** programmers view. 10

**PVT** programmers view with timing. 10

**RTC** run-to completion. 46

**RTL** register transfer level. iii

**RTOS** real time operating services. 45

**SAL** software abstraction level. 93

**SDL** specification and description language. 21

**SPU** symbolic program unit. 17

**SW** software. 3



**TLM** transaction-level modeling. 9

**TTI** transmission time interval. 26

**UPF** unified power format. 15

**VCD** value change dump. 25

**VPU** virtual processing unit. 23

**WCET** worst-case execution time. 20

## Publications of the Author

- [1] **Ayub, Muhammad Mudussir** and Kreupl, Franz. "A Modular and Distributed Setup for Power and Performance Analysis of Multi-Processor System-on-Chip at Electronic System Level". In: *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*, 2020, 1–8, IEEE.
- [2] **Ayub, Muhammad Mudussir** and Ahmadzay, Habibullah and Eckmüller, Josef and Kreupl, Franz. "Electronic System Level Power and Performance Analysis for Multi-Processor-System-on-Chip". In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, 1–2, IEEE.
- [3] Frascolla, Valerio and Sue, Jonathan Ah and **Ayub, Muhammad Mudussir** and Miesniak, Krzysztof and Hasholzner, Ralph and Englisch, Jürgen and Ben-Ameur, Amal. "Cross-layer optimization in terminals". In: *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, 802–806, IEEE.
- [4] **Ayub, Muhammad Mudussir** and Josef, Eckmuller and Francois, Philipp. "Intel Docea Power Analyser Comparison with KV2, and Future of Power Modeling". In: *Intel® Power Summit (IPS)*, 2018, Intel® Internal Conference.
- [5] **Ayub, Muhammad Mudussir** and Jürgen, Brück. "Pre-Silicon Power and Performance Power Analysis with Physical Data Emulation (PDE) Framework". In: *Intel® Software Conference (ISC)*, 2017, Intel® Internal Conference.

## **Erklärung**

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellenachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

München, date

Name