LEHRSTUHL FÜR INTEGRIERTE SCHALTUNGEN
Univ.-Prof. Dr.-Ing. I. Ruge, em.

PERFORMANCE ESTIMATION FOR THE DESIGN SPACE EXPLORATION OF SYSTEM-ON-CHIP SOLUTIONS

Nuria Pazos Escudero

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Georg Färber

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. Ingolf Ruge, em.
2. Univ.-Prof. Dr. rer. nat. Wolfgang Rosenstiel,
   Eberhard-Karls-Universität Tübingen

Die Dissertation wurde am 30.04.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 29.07.2003 angenommen.

Acknowledgements

First of all, I want to thank Prof. Ingolf Ruge who encouraged and enabled me to pursue this Ph.D. thesis at his institute. Furthermore, I want to thank the members of the dissertation committee, Prof. Wolfgang Rosenstiel and Prof. Georg Färber for the time and effort they gave to this cause.

Specially, I thank my tutor and leader of the networking group at the institute Thomas Wild for motivating me and for his patience. The technical discussions with him have contributed significantly to my scientific work. It has been a pleasure and an honour to work with him. At the same time, I greatly thank Dr. Walter Stechele his daily trust on me.

My very special thanks are directed to Nabil Ouerhani. Without his substantial and daily technical as well as private support and encouragement I would not have been able to reach this goal.

To those individuals who spent their time proof-reading my papers and the manuscript of this work, I am particularly grateful. Their suggestions and corrections were very helpful in clarifying various points that might otherwise have been mistreated or omitted. A special thank goes again to Thomas Wild and to Paul Zuber, to my colleagues Axel Hof, Hubert Mooshofer and Jürgen Foag, to my friends Dr. Magdalena Rafecas, Dr. Javier Bracamonte, Markus Kuhn, Raymond Korhon and, finally, to Ilka Nahmmacher, who read carefully the final version.

I would also like to thank other colleagues I had the pleasure to interact with: Stephan Herrmann, Torsten Mahnke, Raul Medina, Ulrich Niedermeier and Armin Windschiegl. This also includes my former Master's students Susana Martin and Jorge Juan Ramos and the internship student Amit Chaudhari, whose contributions have been particularly valuable.

Concerning my industrial experience, I would like to thank our project partners from Siemens ICN, Karl Schrodi and Dr. Thomas Theimer, for showing me the huge amount of industrial applications the world of networking comprises.

I owe many thanks to the system administrators at the institute Wolfgang Kothz and Stephan Herrmann, who always provided quick help for my infinite computer problems. I also want to express my thanks to Verena Draga, Gabi Spöhrle and Doris Zeller for their administrative help.

A particular acknowledgement is directed to the persons who pushed the start of my adventure in Germany, Dr. Teresa Riesgo, Dr. Yago Torroja, Dr. Eduardo de la Torre and Prof. Javier Uceda from the Universidad Politecnica de Madrid.

Last but not least, I cannot forget to thank my dear family their unconditional support and encouragement and my friends all over the world for being there. It is not easy to be far away from home, but they have made it possible that I feel close to them during the last years. I owe special thanks to my parents, Francisco and Aurora, and my sister Montse and my brother Francisco for everything they have done for me and for their blind support. This work is dedicated to my father, who injected me the feeling for engineering.

Kurzfassung

Die steigende Komplexität aktueller VLSI–Systeme und die Erhöhung des Marktdrucks für diese Produkte zwingen die Entwickler, sich zu höheren Abstraktionsebenen zu bewegen und den Entwurfsablauf mit ausführbaren Spezifikationen zu beginnen. Auf der System–Ebene wird sowohl der Ablauf zur Auswahl der optimalen Architektur als auch die Partitionierung der Funktionalitäten beträchtlich beschleunigt.

Die Entwurfsraum–Untersuchung ist eine der wichtigsten Aufgaben innerhalb des Entwurfs auf System–Ebene. Sie versucht, die beste Architektur–Partitionierungsalternative zu finden, d.h. die optimale Partitionierung der Systemfunktionalitäten zwischen Hardware und Software und gleichzeitig die richtigen Architektur-Ressourcen, welche die Anforderungen der Spezifikation erfüllen. Die Performance–Abschätzung spielt eine wichtige Rolle innerhalb der Entwurfsraum–Untersuchung, um die Entwurfsalternativen einzuordnen. Performance–Abschätzung wird mit der Bestimmung der Auslastung und des Durchsatzes der Ressourcen gekennzeichnet.

Die vorliegende Arbeit stellt eine neuartige Methode für die Performance–Abschätzung auf der Systemebene vor, die auf der Beschreibungssprache SystemC beruht. Wesentlich bei diesem Ansatz ist, dass kein aufwendiger Umbau des Strukturmodells erforderlich ist, um verschiedene Partitionierungsalternativen zwischen Hardware und Software zu untersuchen. Folglich ist der Modellierungsaufwand bei der Verwendung der vorgeschlagenen Methode wesentlich niedriger, als wenn ein Strukturmodell der Zielarchitektur auf einer niedrigeren Abstraktionsebene verwendet würde. Gleichzeitig beschleunigt diese Methode die Simulation und verbessert die Genauigkeit der Ergebnisse bisheriger Ansätze. Diese Eigenschaften ermöglichen eine schnelle und einfache Exploration von mehreren Alternativen bei einer Entwurfsraum–Untersuchung.

Die vorgeschlagene Performance–Abschätzungsmethode ist dafür gedacht, innerhalb eines Hardware–Software Co–Design Ablaufs integriert zu werden, bei dem mehrere Zielarchitekturen untersucht werden und eine einfache Zuordnung der Funktionen zur Zielarchitektur möglich sein sollte.

Der Industriestandard SystemC wurde als Beschreibungssprache für die Implementierung der Methode gewählt. Es handelt sich dabei um eine neue Verwendung dieser Sprache, wobei die Systemfunktionalitäten und die Eigenschaften der Zielarchitektur in Form eines Graphen funktional beschrieben sind. Frühere Ansätze haben SystemC angewendet, um die Systemfunktionalitäten und die Zielarchitektur strukturell zu beschreiben.

Die Methode beruht auf drei Modellen, dem funktionalen, dem architekturalen und dem Kommunikationsmodell, die aufeinander aufbauen. Die Spezifikationsfunktionen sind in Form eines Prozessgraphen beschrieben, der mit den Eigenschaften der Zielarchitektur weiter gekennzeichnet ist. Diese Annotierungsinformation bezieht sich auf die Abbildung und das Scheduling der Funktionen zu Prozessierungseinheiten und auf die Betrachtung der Systemkommunikation und der Lösung von Zugriffskonflikten. Der Implementierungsablauf für die Erzeugung der drei Modelle ist anhand entsprechender Konfigurationsdateien automatisiert.

Für die Verifikation der vorgeschlagenen Methode wurde eine flexible Co–Simulationsplattform entwickelt, die ein strukturelles Modell der Zielarchitektur darstellt. Ein Zyklen–genaues Modell der Plattform wurde implementiert. Es dient als Referenz für die Bewertung der Ergebnisse der vorgeschlagenen Methode. Auf Basis der Ergebnisse der Plattform wurden die Simulationszeit und der Ausgangsdurchsatz der neuen Methode verifiziert.

Die Methode wurde auf ein Hardware–Software System für die Bearbeitung von Netzwerk–Protokollen erfolgreich angewendet. Eine System–on–Chip Architektur, die mehrere eingebettete Prozessoren, die ihrerseits Multi-Threading unterstützen, und dedizierte Hardware-Beschleuniger beinhaltet, wurde als Zielarchitektur für die Zuordnung der Funktionen ausgewählt. Es wurde gezeigt, dass die neue Methode eine um rund 70% schnellere Simulation als eine Zyklen–genaue Simulation erreicht, während die Genauigkeit der Ergebnisse innerhalb einer akzeptablen Toleranz bleibt (etwa 1,5% Abweichung). Wesentlich dabei ist, dass die Untersuchung von mehreren Partitionierungsalternativen beschleunigt wird. Dies ist durch die Verwendung eines funktionalen Graphenmodells, das zusammen mit Architekturdaten das Verhalten des Systems nachbildet, ermöglicht. Jedes Mal, wenn eine neue Zuordnungsalternative untersucht wird, ist es lediglich nötig, die Konfigurationsinformation für die neue Alternative zu ändern. Auf diese Weise kann das Verfahren zur automatischen Untersuchung einer grossen Anzahl an Architekturalternativen eingesetzt werden.

Abstract

The increasing complexity of current VLSI systems and the increase of time-to-market pressure for these products are forcing the developers to move to higher levels of abstraction and to design products from executable specifications. At the system level, the process towards the selection of the optimal target architecture as well as the partitioning of the functionalities can be considerably accelerated.

Design space exploration is one of the most important tasks of system level design. It tries to find the best architecture–partition alternative, meaning the optimal partition of the system functionalities between hardware and software and, at the same time, the right architecture resources, which meet the requirements of the specification. Performance estimation plays a decisive role inside the design space exploration for ranking the design alternatives. Performance estimation is characterised by measuring utilisation and throughput of the resources.

The present thesis introduces a novel system level performance estimation methodology based on the system level language SystemC. Especially significant is that no costly rebuilding of the structure of the model is necessary in order to explore different hardware–software partition alternatives. Consequently, the modelling effort when applying the proposed method is considerably lower compared to building up a structural model of the target architecture at a lower level of abstraction. At the same time, this methodology speeds up the simulation and enhances the accuracy of the results of existing approaches. All of these characteristics allow for fast and easy exploration of several alternatives during design space exploration.

The performance estimation methodology proposed is intended to be integrated in a hardware–software co-design procedure, where several target architectures have to be explored and an easy re-mapping of the functions onto the target architecture should be possible.

The industry standard language SystemC has been chosen as the description language for the implementation of the methodology. It is noteworthy that this is a new application of this language, where the system functionalities and the characteristics of the target architecture are described functionally in terms of a graph. Existing approaches have applied SystemC to describe the system functionalities and the target architecture in a structural way.

The method is based on three models, the Functional, Architectural and Communication model, each of which is built on top of the previous one. The functions of

the specification are described in terms of a process graph, which is further annotated with the characteristics of the target architecture. This annotation information is related to the mapping and scheduling of the functions to the processing units and to the consideration of the system communication and the solution of access conflicts. The implementation procedure towards the creation of these three models has been automated by means of corresponding configuration files.

For the verification of the proposed method, a flexible hardware–software co-simulation platform is developed, which represents a structural model of the target architecture. A cycle accurate model of this platform is implemented, which serves as a reference for the evaluation of the results of the proposed methodology. Taking the results of the platform as a basis for the comparison, the simulation runtime and the output throughput of the new methodology are verified.

The methodology has been successfully applied to a hardware–software system for networking protocol processing. A System-on-Chip architecture, which comprises multiple embedded processors, each of it with multi-threading support and dedicated hardware accelerators, is taken as target architecture for the mapping of the functionalities. It is demonstrated that the new methodology achieves about 70% faster simulations than a cycle accurate simulation, while maintaining the accuracy within an acceptable tolerance (around 1.5%). Significantly by the proposed methodology is that it considerably accelerates the exploration of several partition alternatives by applying a functional graph model, which, together with architecture information, reproduces the behaviour of the system. Each time a new partition alternative is explored, it is only necessary to change the configuration information for the new alternative to be tested. In this manner the procedure can be employed for an automatic exploration of a large amount of architecture alternatives.

Contents

Acknowledgements ii

Kurzfassung iii

Abstract v

Index of Contents vii

List of Tables xiii

List of Figures xv

List of Acronyms xix

1 Introduction 1

1.1 Motivation 2

1.2 Novelty Aspects 3

1.3 Procedure 4

1.4 Structure of the Work 5

2 State of the Art 7

2.1 Chapter Introduction 7

2.2 System Level Design. Languages and Methodologies 8

2.2.1 Classical Hardware–Software Co-Design Procedure 8
2.2.2 System Level Design Procedure 11
2.2.3 System Level Languages 13
2.2.3.1 C/C++-Based Languages 15
2.2.3.2 Domain-Specific System Level Languages 17
2.2.3.3 Extensions to Hardware Design Languages 17
2.2.4 Computer-Aided Co-Design Methods and Tools 18
2.2.4.1 Classification of Co-Design Approaches. 19
2.3 Performance Estimation Approaches 25
2.3.1 Block Level Performance Estimation 25
2.3.1.1 Software Performance Estimation Techniques 25
2.3.1.1.1 Cycle-Accurate Performance Model. 26
2.3.1.1.2 Timing Annotation of the Control Flow Graph (CFG). 26
2.3.1.1.3 Original C-Code Annotation. 26
2.3.1.1.4 Software Estimation by Means of Linear Equations. 27
2.3.1.2 Hardware Performance Estimation Techniques 27
2.3.1.2.1 Local Scheduling Methods 28
2.3.1.2.2 Global Scheduling Methods 28
2.3.2 System Level Performance Estimation 29
2.3.2.1 Simulation-Based Approaches 29
2.3.2.2 Trace-Based Performance Analysis Strategies 30
2.3.2.3 Static Performance Estimation Methods 31
2.3.2.4 Analytical Performance Estimation Methods 32
2.4 VLSI Architectures for Networking Applications 34
2.4.1 Network Infrastructure 34

2.4.1.1 Computer Networks 34
2.4.1.1.1 Communication Layers in a TCP/IP Network. 35
2.4.1.1.2 Communication Processing Tasks. 36
2.4.1.1.3 Classification of Computer Networks. 38
2.4.1.2 Network Equipment 38
2.4.2 VLSI Networking Architectures Design 39
2.4.2.1 Evolution of VLSI Networking Architectures 39
2.4.2.2 Design Trade-Offs 41
2.4.3 Network Processors 41
2.4.3.1 Common Characteristics 41
2.4.3.2 Main Attributes 42

3 Performance Estimation for Design Space Exploration 45

3.1 Chapter Introduction 45
3.2 Design Space Exploration 46
3.3 Performance Estimation 49
3.3.1 Performance Estimation Requirements 50
3.3.2 Categorisation of Performance Estimation Approaches 51
3.3.2.1 Analytical Modelling and Evaluation 52
3.3.2.2 Analytical Modelling and Evaluation by Means of Simulation 52
3.3.2.3 Instruction Level Models 53
3.3.2.4 Cycle-Accurate Models 53
3.3.2.5 RTL and Logic Level Models 53
3.4 Comparison of Performance Estimation Approaches 54
3.4.1 Advantages and Disadvantages 54
3.4.2 Support of Co-Design Procedures 55

3.4.3 Need for Improvements 56

3.5 Integration in the Design Flow 57

3.5.1 Inputs Required by the Performance Estimation 58

3.5.2 Boundary Conditions 59

4 System Level Performance Estimation for Multi-Processing, Multi-Threading SoC Architectures 63

4.1 Chapter Introduction 63

4.2 Fundamentals of the Methodology 63

4.3 Modelling Approach 65

4.3.1 Functional Annotated SystemC Conditional Synchronisation Graph (Functional ASCSG) 66

4.3.2 Architectural Annotated SystemC Conditional Synchronisation Graph (Architectural ASCSG) 67

4.3.3 Communication Annotated SystemC Conditional Synchronisation Graph (Communication ASCSG) 68

4.4 Evaluation Approach 70

4.5 System Performance Estimation Scheme 71

4.5.1 Inputs for the Modelling of the ASCSG 71

4.5.1.1 Sequence of Nodes 73

4.5.1.2 Convergence of Paths 73

4.5.1.3 Flexible Mapping 74

4.5.1.4 Multiple Instances of *Computation Nodes* 74

4.5.2 Outputs of the Evaluation of the ASCSG 75

4.5.2.1 Functional Validation 75

4.5.2.2 Processing of Nodes 75

4.5.2.3 Tracking and Monitoring 76

5 Implementation 77

5.1 Chapter Introduction 77
5.2 Implementation of the ASCSG 77
5.2.1 Creation of the Functional Model 78
5.2.2 Creation of the Architectural Model 79
5.2.3 Creation of the Communication Model 81
5.3 Automation Approach 85
5.3.1 Data Structure for the Implementation 85
5.3.2 Configuration Files 88

6 Methodology Verification 93

6.1 Chapter Introduction 93
6.2 Hardware–Software Co-Simulation Techniques 94
6.2.1 Techniques Requiring Processor Models 95
6.2.2 Techniques not Requiring Processor Models 96
6.3 Target Platform Architecture 97
6.4 Modelling Approach 98
6.4.1 Skeleton of the Co-Simulation Platform 99
6.4.2 Functional Units 101
6.4.2.1 Embedded RISC Processor 101
6.4.2.2 Hardware Blocks – Accelerators 104
6.4.2.3 Shared Memories 104
6.4.3 Communication Structure 105
6.4.3.1 Command Bus 106
6.4.3.2 Data Bus 107

7 Results and Evaluation 109

7.1 Chapter Introduction 109
7.2 Case Study 109

7.2.1 Packet Processor Functionalities 110
7.2.2 TCP/IP Packet Processing Data Flow 111
7.3 Modelling of Architecture–Partition Alternatives 112
7.4 Simulation Results 117
7.4.1 Simulation Environment 117
7.4.2 Simulation Predefinitions 119
7.4.3 Output Throughput 120
7.4.4 Performance Values 120
7.5 Simulation Speed of the ASCSG 124
7.6 Accuracy of the ASCSG 127
7.7 Modelling Effort of the ASCSG 127
7.8 Comparison with Other Performance Estimation Methods 130

8 Summary and Conclusions 133

Bibliography 145

List of Tables

2.1 Comparison of Hardware–Software Co-Design Approaches 23

3.1 Stirling Numbers of the Second Kind 49

3.2 Comparison of Architecture Simulators 54

5.1 Possible Mapping Execution Times 79

7.1 Architecture–Partition Alternatives Explored 119

7.2 Parameter Processing Units 119

7.3 Parameter Communication Media 120

7.4 Values Block Level Performance Estimation 120

7.5 Output Throughput (packets/s) 121

7.6 Simulation Speed 126

7.7 Simulation Accuracy 128

7.8 Comparison Performance Estimation Approaches 131

List of Figures

2.1 Classical Hardware–Software Co-Design Procedure 9

2.2 System Level Design Flow 12

2.3 Classification of Co-Design Approaches 20

2.4 TCP/IP Conceptual Layers 36

2.5 Communication Processing Tasks 37

2.6 VLSI Networking Architectures 40

3.1 Abstraction Levels for Models 46

3.2 Design Space Exploration 48

3.3 Performance Estimation Abstraction Levels 52

3.4 Integration Performance Estimation Process in the Design Flow 57

3.5 Target Architecture 61

4.1 Estimation of Performance 64

4.2 Functional ASCSG 66

4.3 Architectural ASCSG 67

4.4 Shared *Computation Node* 68

4.5 Communication ASCSG 69

4.6 System Performance Estimation Scheme 72

4.7 Sequence of Nodes 73

4.8 Convergence of Nodes 73
4.9 Mapping onto PE/Thread–AC 74
4.10 Multiple Instances of *Computation Nodes* 75
4.11 Functional Validation 76

5.1 ASCSG Implementation Procedure 78
5.2 Access to a Shared *Computation Node* 81
5.3 Access to a Shared Communication Medium 82
5.4 Access to a Shared Communication Medium from/to a Hierarchical Channel 83
5.5 Context Switch and Context Event Notification in a Multi-Threading Processor 84
5.6 *Computation Node* Data Structure 86
5.7 *Access Node* / *Shared Element* Data Structure 87
5.8 *Communication Node* / Arbiter Data Structure 88
5.9 Performance Estimation by Means of Configuration Information 91

6.1 Target Platform Architecture 98
6.2 Bus-Based Co-Simulation Platform 100
6.3 Embedded RISC Processor 102
6.4 Multi-Threading Modelling 103
6.5 FSM of Context Event Arbiter 103
6.6 Internal Architecture of a Hardware Block 105
6.7 Internal Architecture of a Shared Memory 106
6.8 Command Bus 106
6.9 Data Bus 106

7.1 Case Study: Input Packet Processing 111
7.2 Case Study: Functional ASCSG 113

7.3 First Alternative: Two PEs and One AC 114
7.4 Target Architecture First Alternative 115
7.5 Second Alternative: Two PEs and Two ACs 116
7.6 Target Architecture Second Alternative 117
7.7 Third Alternative: One Multi-Threading PE and One AC 118
7.8 Queue Command Bus 122
7.9 Queue Data Bus 122
7.10 Queue Classify Accelerator 122
7.11 Queue LPM Accelerator 122
7.12 Queue Events PE_0 123
7.13 Queue Events PE_1 123
7.14 Delay Write Request Command Bus; PEs 123
7.15 Delay Write Request Command Bus; Threads 123
7.16 Delay Write Request Data Bus; PEs 124
7.17 Delay Write Request Data Bus; Threads 124
7.18 Delay Write Request Classify; PEs 124
7.19 Delay Write Request Classify; Threads 124
7.20 Delay Write Request LPM; PEs 125
7.21 Delay Write Request LPM; Threads 125
7.22 Delay Wait Event PE_0 125
7.23 Delay Wait Event PE_1 125
7.24 Modelling Effort 129

List of Acronyms

- AC: Accelerator
- ALU: Arithmetic Logic Unit
- ANSI: American National Standards Institute
- ARPANET: Advanced Research Project Agency NETwork
- AS: Autonomous System
- ASAP: As Soon As Possible scheduling algorithm
- ASCSG: Annotated SystemC Control-Synchronisation Graph
- ASIC: Application Specific Integrated Circuit
- ASSP: Application Specific Standard Products
- BFM: Bus Functional Model
- CAD: Computer-Aided Design
- CAG: Communication Analysis Graph
- CBA: Command Bus Arbiter
- CDFG: Control Data Flow Graph
- CEA: Context Event Arbiter
- CFG: Control Flow Graph
- CFSM: Co-design Finite State Machine
- Cme: external Communication node
- Cmi: internal Communication node

- COSYMA: COSYnthesis of eMbedded microArchitectures
- CPG: Conditional Process Graph
- CPI: Communications Programming Interfaces
- CP: ComPutation node
- CPU: Central Processing Unit
- CSR: Control and Status Registers
- DiffServ: Differentiated Services
- DSP: Digital Signal Processor
- EDA: Electronic Design Automation
- ESC: Extended SystemC library
- FDDI: Fiber Distributed Data Interface
- FPGA: Field Programmable Gate Array
- FSM: Finite State Machine
- FTP: File Transport Protocol
- GNU: GNU's Not Unix
- GPR: General Purpose Register
- HDL: Hardware Description Language
- HDLC: High level Data Link Protocol
- HTTP: Hyper Text Transfer Protocol
- IP: Intellectual Property
- IP v.6: Internet Protocol version 6
- ISO: International Organisation for Standardisation
- ISP: Internet Service Provider
- ISS: Instruction Set Simulator
- ITRS: International Technology Roadmap for Semiconductors

- LAN: Local Area Network
- MAC: Medium Access Control
- MAN: Metropolitan Area Network
- MIPS: Million Instructions Per Second
- M: Memory unit
- OC: Optical Carrier
- ODETTE: Object-oriented co-DEsign and functional Test TEchniques
- OSI: Open Systems Interconnection
- PE: Processing Element
- POOSL: Parallel Object-Oriented Specification Language
- PNI: Programmable microprocessors on Network Interfaces
- PU: Processing Unit
- QoS: Quality of Service
- RFIFO: Receive First In First Out
- RISC: Reduced Instruction Set Computer
- RTL: Register Transfer Level
- RTOS: Real Time Operative System
- SDL: Specification and Description Language
- SDRAM: Synchronous Dynamic Random Access Memory
- SIA: Semiconductor Industry Association
- SIR: System Intermediate Representation
- SLA: Service Level Agreement
- SLDL: System Level Design Language
- SoC: System-on-Chip
- SONET: Synchronous Optical Network

- SPADE: System level Performance Analysis and Design Space Exploration
- SPI: System Property Intervals
- SRAM: Static Random Access Memory
- TCP: Transport Control Protocol
- UML: Unified Modelling Language
- VC: Virtual Component
- VHDL: VHSIC Hardware Description Language
- VHSIC: Very High Speed Integrated Circuits
- VLSI: Very Large System Integration
- WAN: Wide Area Network

Chapter 1

Introduction

At present, the complexity of VLSI integrated systems is growing exponentially. This is partially supported by the advances in silicon processing technology, which enable integration of ever more complex functions on a single chip. These so-called Systems-on-Chip (SoC) contain dedicated hardware components, programmable processors, memories, etc., requiring not only the design of digital and analogue hardware but also the design of embedded software. Nevertheless, the well known chart published by the Semiconductor Industry Association (SIA) [1] predicts an increase in design complexity of about 58% per year while designer productivity increase stays at 21% per year. These predictions demand new methodologies and tools that will allow significant productivity improvements beyond the present trend.

Moreover, increasingly more complex circuits have to be designed in less time and with greater guarantees of success in order to compete in a market that is becoming more and more aggressive. This demanding environment is forcing fundamental changes in the way VLSI systems are designed. The use of predefined Intellectual Property (IP) blocks for System-on-Chip (SoC) design has become essential in order to build the required complexity in a short time-to-market. The designers see current design tools and methodologies as inadequate for developing million gate ASICs from scratch. Tools are not providing the productivity gains required to keep pace with the increasing gate counts available from deep submicron technology. Design reuse, i.e. the use of pre-designed and pre-verified cores, is the most promising opportunity to bridge the gap between available gate count and designer productivity.

A further interesting consequence of this increase of design complexity and increase of time-to-market pressure is that they are forcing companies to move to higher and higher levels of abstractions and to design products from executable system specifications. In the International Technology Roadmap for Semiconductors (ITRS) [2] 2001 is written

that “the cost of design is the greatest threat to the continuation of the semiconductor roadmap”. Although design reuse and implementation tools contribute to decrease the design cost, in the future the ITRS claims for “intelligent test-benches” and “embedded system level methodologies”.

The design of network equipment constitutes a good example of complex systems with an enormous growing perspective. Nowadays, the network equipment vendors are racing to provide the new converged voice–video–data communication infrastructure. Furthermore, conservative estimates for aggregate bandwidth on the Internet backbone indicate a doubling each year for the past ten years. One consequence of this growth is the demand for greater performance, flexibility, reliability and cost effectiveness in the routers and switches which control the flow of data through the network.

1.1 Motivation

The new challenges concerning speed and flexibility pursued by emerging networking architectures introduce in their design a scenario of multiple alternatives. The selection of the optimal target architecture as well as the partitioning of the functionalities that fulfills the constraints should start at a high-level of abstraction, i.e. at system level, where different trade-offs can be fixed. Moreover, from the user’s point of view of a certain architecture, the best mapping of the functionalities onto the fixed architecture can be facilitated if a model of the system at a high level of abstraction is provided.

In a traditional design methodology, hardware and software design takes place in isolation with the hardware being integrated with the software after the hardware is fabricated. At system level, engineers are reconsidering how designs are specified, partitioned and verified. In the actual complex systems, the software is programmed in C/C++ and the corresponding hardware is developed in description languages such as VHDL and Verilog. Thus, it is becoming common that problems arise from the use of different design languages and incompatible tools. Errors that cannot be fixed in software lead to costly re-fabrication and adversely affect time-to-market. To avoid costly silicon re-spins and improve time-to-market, the design methodology has to change, so that the hardware and the software are integrated earlier in the design cycle that leads to the so-called system level design.

At system level, a huger scenario of possible alternatives have to be explored and, therefore, the new embedded system level methodologies have to improve the support of fast and flexible design space exploration procedures. This can be achieved by applying fast and flexible performance estimation methods together with exploration strategies, in order to rank the design alternatives. Performance estimation is characterised by

measuring utilisation and throughput of the resources.

In traditional system design methodologies the target architecture is predefined. The designer holds the design choices in terms of hardware–software partitioning, which opens a huge space for exploration. During the partitioning phase, the functionality captured by the specification is distributed among the allocated system components. If the designer additionally wants to explore various architecture choices, the design space becomes more dense. This creates the need for a fast exploration process, where the performance evaluation of a potential solution plays a decisive role for reducing the design alternatives. If the exploration is performed at a higher level of abstraction, a huger scenario of low level alternatives is covered. Afterwards, at lower levels of abstraction, only the best alternatives will be further evaluated.

It can be summarised that the acceleration of the exploration process requires a fast performance estimation of the functionalities without losing accuracy. A compromise between accuracy and computation time determines the feasibility of the estimation techniques. Moreover, the time consumption of the whole design process could be drastically reduced if high level estimation methodologies would be performed before taking major design decisions.

A further related issue to be supported by the new system level methodologies is the description language. If a design consists of hardware and software, the modelling language should be C/C++ since standard processors come only with C/C++ compilers. Unfortunately, the C/C++ language was developed for describing software and not hardware. It is missing basic constructs for expressing hardware concurrency and communication among components. Therefore, a language is needed that can be compiled with standard compilers and that is capable of modelling hardware and software on different levels of abstraction.

1.2 Novelty Aspects

The present work introduces a novel system level performance estimation methodology based on the system level language SystemC. The rebuilding effort is considerably lower when applying the proposed methodology compared to building up a structural model of the target architecture at a lower level of abstraction. It accelerates the exploration of several partitioning alternatives of a system specification onto a target architecture. This is achieved by applying a graph, whose structure does not have to be re-constructed each time a new alternative is explored. Only the information concerning the new architecture–partition configuration to be tested has to be provided.

The performance estimation methodology proposed is intended to be integrated in a

hardware–software co-design procedure, where several target architectures have to be explored and an easy re-mapping of the functions onto the target architecture should be possible. The current methodology achieves a fast simulation runtime while maintaining the accuracy within an acceptable tolerance. The latter is achieved by taking into account the communication loads among components when the partitioning decision is taken.

The goals pursued by the new methodology are achieved by means of a two-step procedure. First, the system is modelled in terms of a graph, where the relevant information concerning the target architecture is annotated and a mechanism to solve resource contentions is added. This functional modelling accelerates the exploration of further alternatives without requiring a re-building of the structure of the graph if the hardware structure is changed. Second, the evaluation is performed by means of simulation at system level. It allows a more detailed analysis than analytical evaluation. And, at the same time, the simulation runs faster than a simulation at a lower level of abstraction.

For the modelling of the graph, the industry standard system level language SystemC has been chosen. It is noteworthy that this is a new application of this language, which supports the implementation of the whole methodology. SystemC characteristics such as the modelling of time, reactivity and concurrency, and the assistance it provides in evaluating resource contentions, make this language especially suitable for the implementation of the methodology. Existing approaches have applied SystemC to describe the system functionalities and the target architecture in a structural way.

The proposed method is oriented to support the design of multi-processing and multi-threading architectures for networking applications. These architectures contain, mainly, multiple RISC embedded processors, powerful co-processors to accelerate some costly functions and memory and interface blocks. Different communication resources connect all building blocks with each other. Very often a bus-based communication is found. An important characteristic of RISC embedded processors is its multi-threading hardware support to avoid idle times when waiting on results from other resources. The modelling and evaluation support of this characteristic also constitutes a novelty aspect of the proposed method, which has not been previously covered by other performance estimation approaches.

1.3 Procedure

An introduction to the problems behind design space exploration of the most recent systems is needed for understanding the requirements imposed on the performance

estimation methods. This theoretic study, together with an exploration of the existing techniques, will show the shortcomings and need for improvement in this area. In particular, the current work concentrates on the reduction of the re-building effort, which is quite high in previous approaches based on a structural model of the system target architecture.

As input for the work, a graph representation for modelling the system functionalities is selected. This graph is further annotated with information concerning the target architecture and the selected partitioning of the functionalities onto the available resources. Furthermore, the required mechanisms to solve resource contentions are added to the graph. Then, the evaluation of the graph by means of simulation delivers the performance estimation values. In order to verify the new approach in terms of simulation time and accuracy, a comparison of the simulation results with a cycle-accurate model is made for a specific example from the networking world. Finally, the results achieved by the proposed system level performance estimation methodology are compared with the results of other approaches. Especially attention is paid to the abstraction level, modelling effort, simulation time and accuracy.

1.4 Structure of the Work

The remainder of this work is organised as follows: Chapter 2 presents the state of the art in the related areas of system level design, performance estimation techniques as well as in the main application scenario, VLSI networking architectures. In Chapter 3 the role of performance estimation within design space exploration is presented and the problems are explained. Furthermore, the disadvantages and shortcomings of the previous approaches are shown, which constitute the main motivations for the development of the proposed method. Then, the integration of the proposed performance estimation method into the hardware–software design flow is depicted and the boundary conditions and assumptions taken are explained. Chapter 4 describes the modelling approach. This encompasses the Functional, Architectural and Communication models. Moreover, the system performance estimation scheme is illustrated by means of selective examples from the networking world. The steps towards the implementation of the methodology are explained in Chapter 5. Chapter 6 depicts the configurable cycle-accurate hardware–software co-simulation platform built for verification purposes. Chapter 7 compares the performance results achieved by the proposed methodology to the outcomes of the cycle-accurate platform. The results are evaluated and, further on, the procedure of the new method is compared with other performance estimation techniques. The work ends with a summary and the conclusions.

Chapter 2

State of the Art

2.1 Chapter Introduction

In order to position the topic of this thesis, system performance estimation, as part of the system level design, a comparison between the classical hardware–software co-design procedure and the system level design is performed first. Later on, the most recent and relevant system level languages are classified and briefly explained. Subsequently, some well known computer aided co-design methods and tools developed at different universities and companies are introduced.

The estimation of the performance achieved by the architecture–partition alternative under study plays an important role in this system level design paradigm. Several approaches dealing with this issue at different abstraction levels are then shown and their advantages and disadvantages discussed.

An introduction to the application scenario, VLSI networking architectures, is finally carried out. An overview of the evolution of such architectures will show the requirements when pursuing the design of such complex systems. It demands the exploration of different trade-offs at system level, where the designer has more freedom to achieve a reasonable compromise.

2.2 System Level Design. Languages and Methodologies

Nowadays, the increase in complexity of the current VLSI designs has forced an evolution in the nature of the systems under design. The definition of a system has moved from the system-on-board to the System-on-Chip (SoC). In the past, the systems were composed of discrete parts such as microprocessors, memory chips, analogue devices and application-specific integrated circuits (ASICs). In contrast, the modern System-on-Chip may well contain one or more processors including both microcontrollers and digital signal processors (DSPs) or specialised processors. Moreover, it includes on-chip memory and peripheral control devices, linked together by an on-chip communication network.

System design of embedded systems is often perceived as a process going from a functional specification of a system, through a number of refinement or transformation steps to an architecture, and from there, to a final implementation ([3]).

The increase in complexity and reduction of time-to-market requirements for current embedded systems has resulted in the fact that the first phases in the design process, i.e. design specification, allocation of the architecture and mapping of the functions onto the target architecture, are no longer carried out from scratch, but rather with the help of abstract models. This Section firstly depicts the classical hardware–software co-design procedure and, later on, demonstrates why this does not meet the design requirements introduced by today's complex systems. The whole procedure has thus to start at a higher level of abstraction, i.e. at system level, where the designer has more freedom to explore the trade-offs and detect the bottlenecks. System level design is mainly characterised by both, high-level of abstraction and exploration, where the latter attribute is a consequence of the former one.

2.2.1 Classical Hardware–Software Co-Design Procedure

Hardware–software co-design can be defined as the cooperative design of hardware and software. The flexibility offered by software allows late design changes and simplifies debugging. Furthermore, the reuse of software by porting it to other processors reduces the time-to-market and the design effort. Finally, in most cases, the use of processors is very cheap compared to the development costs of ASICs. However, hardware is always used by the designer when the processors are not able to meet the required performance. This trade-off between hardware and software illustrates the optimisation aspect of the co-design problem.

A typical electronic system development project is divided into four basic phases ([4]): Product concept; design planning; hardware–software design phase; and integration and test phase (co-simulation/emulation), as can be seen in Figure 2.1.

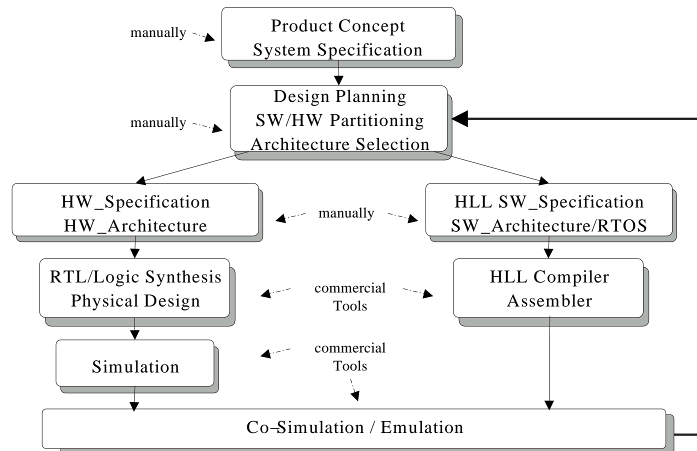

Figure 2.1: Classical Hardware–Software Co-Design Procedure

1. Product Concept. During this phase, the entity being designed is viewed as an overall system rather than as distinct hardware and software components. This phase delivers the specifications for the system behaviour. Additionally, functional requirements and budgets for both hardware and software components of the system are created. These include constraining costs, size, performance, and physical attributes. The system engineer writes a C or C++ model of the system to verify the concepts and algorithms at system level.

2. Design Planning, Hardware–Software Partitioning. Starting from the specification, the design is partitioned into hardware and software parts. This step has to done in a manner by which the system function is preserved. This phase also takes into account the underlying architecture and the resources on which the system is being implemented. Partitioning involves the following major subtasks:

    - Target Architecture Constraints Description: It includes the available resources and their interconnections;
    - Allocation and Binding: The designer schedules certain operations in hardware and others in software and then estimates whether the constraints are satisfied;
    - Cost/Performance Estimation: In order to estimate the overall performance of the system, accurate estimates of the performance of the functions on various resources are required.

3. Hardware–Software Design Phase. During this phase, different teams for the hardware and software part solve their respective problems. The hardware and software design and implementation efforts typically start at the same time and ideally end at the same time. But in practice, the beginning of the software development has to wait until the hardware is finished. The software part is developed using compilers and debugger tools, while for the hardware part, the functionality is described using a hardware design language (HDL). For the parts of the original model to be implemented in software, the model has to be re-written with calls to an RTOS. The conversion of the C/C++ model to be implemented in hardware into the HDL occurs manually. This process is very tedious and error prone. After the conversion took place, the HDL model becomes the focus of development. The C model quickly goes out-of-date as changes are made. Changes are typically made only to the HDL model and will not implemented in the C model.

4. Integration and Test Phase. In theory, integration and test are the final steps where the correctness of the system is checked. In practice, it is the first time that the completed hardware and independently developed software come together as a system. At this time numerous errors appear, as for example, the effects of misinterpretations of interface definitions, out-of-date specifications, poorly communicated changes of the specification between the teams, and ineffective performance modelling. Consequently, one third or more of the total development time is spent in this phase ([5]). As the tools progress and become more friendly to both the hardware and software developers, the overlap between the test phase and the design and implementation phases will increase. Tests that are created to validate the C model functionality cannot be run against the HDL model without conversion. The test suite has to be converted to the HDL environment.

   Developers are often forced to redesign and/or reduce product objectives when integration problems are found. Given the long fabrication cycles and costs associated with re-designing and re-spinning ASICs, the adaptation of the work is frequently performed in software. This is not always the best solution for the end product. But integration and test leads back to the design and implementation phase, and can take about the same amount of time to complete as the original design and implementation. Also in some cases, the first product release will not contain all of the intended software functionality because it has not been possible to start the integration effort earlier in the design.

   Furthermore, two things are necessary before virtual integration and test can be accomplished. The first is the ability to simulate the hardware at speeds sufficient to make software execution a reality. In most cases, this means that the overall simulation performance must be increased by a factor of at least 1000 over the

current execution speeds for hardware-oriented simulation products. The model is simulated and verified with an RTOS emulator. Some parts of the original code can be reused, but the change in abstraction from the original model to an RTOS-based model requires significant manual re-coding and verification of changes becomes a problem. The second is the need to bring the debug and development environments for the hardware and software closer together. Simulation waveforms do not provide a natural way for a software engineer to debug high-level languages. The original source form for both the software and hardware must be maintained within a single unified debugging environment.

As depicted in Figure 2.1, in a classical hardware–software co-design procedure the first two phases are performed manually, while the last two are partially supported by commercial tools. Moreover, the decision concerning the partition of the functionalities in hardware and software is reached empirically and no exploration takes place. Firstly at the end of the co-design procedure, i.e. in the integration and test phase, come the hardware and software together and, in case the predefined constraints are not met, a new partitioning alternative is tried. This procedure becomes then very tedious in case the test of several alternatives is required.

It can be concluded that the classical co-design approach is no longer feasible for the design of large heterogeneous Systems-on-Chip because quantitative architectural considerations are not taken into account prior to the implementation phase. A move to a level of abstraction beyond the RT level is therefore required — a move to what has been termed system level design.

2.2.2 System Level Design Procedure

One of the most challenging tasks in System-on-Chip design is to map a complex application onto a heterogeneous architecture, assuring that the specified performance and cost requirements are preserved. The required flexibility and performance is best delivered by a heterogeneous system architecture. As a result, the designer faces a huge design space and has to compose a system architecture from various kinds of building blocks in order to meet the constraints of the specific application.

A solution for dealing with this complexity is to exploit hierarchy and to move to higher levels of abstraction ([7]). The level of abstraction is a trade-off with the level of accuracy. A high abstraction level implies low accuracy and vice versa. Nevertheless, a complex embedded system is easier to deal with at the abstract system level than at the detailed gate or transistor level.

The term System Level Design refers to the abstraction level increase in the design

of embedded systems, i.e., from the architecture level to the system level. This is an earlier stage in the design process which was previously carried out manually. Thus, the design process of a new system starts from a highly abstract specification model and ends with a highly accurate implementation model. The advantage of such a top-down approach is that all necessary design decisions can be made at an abstraction level where the irrelevant details are left out of the model. At system level, the designer has more freedom.

Figure 2.2 illustrates the system level design flow. Further on, the different phases are explained in detail.

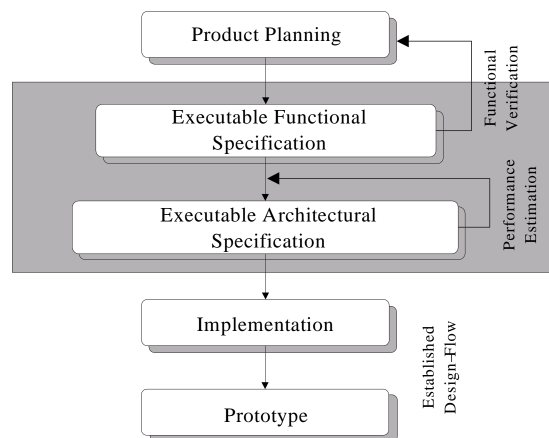

Figure 2.2: System Level Design Flow

The system level design process begins with the product planning, where there is still a close interaction between the customer and the designer. They both reach an agreement concerning the system functionalities, which is written in terms of a specification model that can be simulated (executable functional specification). It forms the input to architecture exploration and therefore defines the basis for all exploration and synthesis. The specification model is purely functional, that is, it is free of any implementation details and of any notion of time. The model executes in zero simulation time.

The specification model describes the required behaviour. There are different flavours of models, characterised by the abstraction level and the semantics. Some systems are described using finite state machines, either graphically (e.g., Statecharts) or by means of textual languages (e.g., SDL or Esterel), whereas hardware components are often specified using a hardware description language (e.g., VHDL or Verilog). Developing a single language (textual or graphical) that can express all desired features is a difficult task because of the heterogeneity of the system components. Specification and design

frameworks that support multiple and extensible design front-ends is the foreseeable solution. Such frameworks should then support a variety of tools for validation and synthesis of the hardware and software component. The SPI (System Property Intervals) model ([8]) represents a good example of such frameworks. SPI is an internal high-level representation that facilitates global, system level analysis, optimisation and synthesis of heterogeneously specified embedded systems.

Once the functional description has been verified and there are no changes in the description and in the product specification needed, the process enters into the architecture exploration phase. The purpose of this phase is to map the system functionalities represented by the specification model onto the components of the target architecture, which has been previously defined. The steps involved in this phase are as follows: First of all, a target architecture (composed of processing elements and memory units) for the implementation is allocated. Secondly, the partitioning of the functionalities onto the processing elements takes place. This step comprises the mapping of the functionalities onto the processing elements and the scheduling of such functionalities.

This partitioning of the functionalities onto the target architecture opens a large scenario of alternatives, the scope of which grows with the complexity of the system. This architecture exploration phase is supported by an executable architectural specification, which delivers an estimation of the performance achieved by the partition alternative under study. This architecture exploration phase is today partially supported by some tools and methodologies.

The search of the optimal solution that meets the predefined cost constraints is a NP-complete problem ([9]). If a problem is known to be NP-complete, then it is unlikely that a polynomial time algorithm exists for that problem. However, in the design automation field there is an urgent need to solve the problem even if it cannot be solved optimally. As soon as the specified cost constraints are met, the implementation phase starts. From this point on, an established standard design flow follows.

2.2.3 System Level Languages

Fundamentally, a system design language is a notation that embodies semantics for describing a system prior to mapping it onto an architecture. Components must be able to be described without making assumptions about the implementation. The system level design language must have a way to describe the behaviour of the components in the system irrespective of whether they will be mapped to software running on a microprocessor or to application-specific hardware.

It should be possible to use the language to construct a performance model. Such a

model allows exploration of the architecture of the system without exactly specifying what microprocessor or bus specification will be used in the implementation. The decisions can be made later, after the entire system is described and simulated at a very high-level.

The language must also include the ability to incorporate the description of constraints, such as event ordering, timing, dependencies and concurrency, with granularity and scheduling mechanisms.

Understanding what system level design means is crucial for defining the requirements of a system level language. System level languages might provide support for integrating domain-specific descriptions, for more abstract modelling and for verification. There are four major reasons why a system level design language becomes necessary ([3]):

- SoC designs combine hardware and software, not only hardware as in traditional ASIC design. Therefore, there is a need for a language that describes the functionality of both the software and hardware. It is essential that the system is defined first, and the exact implementation (hardware or software) is established later on in the design process;
- SoC designs are increasingly incorporating hardware and software intellectual property (IP) from various sources. All of these sources need to use a common system level design language so that the entire system can be modelled;
- Even hardware-only designs are becoming too complex to simulate in RTL. Simulating the entire design at a higher level provides much faster simulation times and lets the designer test the behaviour of the entire chip before it is produced;
- System level design is also required to develop a virtual prototype of the hardware that the software designers can use to begin software development. The old model of waiting until the hardware is finished to begin software development is not applicable any more.

Currently, many different languages in the area of system design are available. They can be broadly classified with regard to the design phase they support and the application scenario they are focused on. Under the first criterion, the existing design languages can be divided into three classes. The first group contains the system specification and modelling languages for the description of functionality, properties and constraints. Secondly, there are architecture languages for the modelling of an architecture and system IP (Intellectual Property) and VC reuse (Virtual Component). And finally, the last group comprises design command languages for estimation and

validation purposes. A language covering more than one such task is also possible. An important boundary condition when developing a system level language is the fact that there are many different kinds of systems (i.e., different domains with different models of computation). Moreover, within a system, the heterogeneity also has to be taken into account.

In the past, there have been a few attempts at the standardisation in the area of system level languages, but they were not very successful. ACCELLERA ([10]) constitutes a new attempt at standardisation. It was formed in 2000 through the unification of Open Verilog International and VHDL International to focus on identifying new standards, development of standards and formats, and to foster the adoption of new methodologies. ACCELLERA has promoted the development of *Rosetta* ([11]). *Rosetta* is a System Level Design Language (SLDL) developed to address requirement specifications for SoC designs. The mission of the SLDL is to bring together heterogeneous information in a language environment. Specifically, *Rosetta* allows designers to develop and integrate specifications written in multiple semantic models to provide language and semantic support for concurrent engineering of electronic systems. Some current activities around *Rosetta* are standardisation, its integration in the design flow and the definition of *Rosetta* subsets for SystemVerilog, SystemC and other Hardware Design Languages (HDL). The standardisation process is scheduled to be concluded in December 2004.

A first rough classification of the system level design languages can be carried out by differentiating between C/C++ approaches, domain-specific languages and extensions of hardware design languages. Moreover, most of them support the system specification and modelling of functionalities, while only few provide a means of architecture modelling and a support for estimation and validation. A further distinctive criterion is the necessity of no language translation in the further refinement of the specification.

2.2.3.1 C/C++-Based Languages

The C based system level languages offer two main advantages. First of all, they are known worldwide and, secondly, there is no translation necessary for the software part. For the hardware part, methods and automated tools have been appearing in the meantime that will support the translation. The current two big players in the area of C/C++ based approaches are SystemC and SpecC:

- SystemC is a new modelling language based on C++ that is intended to enable system level design and IP exchange at multiple abstraction levels for systems containing both hardware and software components ([12]). The SystemC standard is controlled by a steering group composed of a broad range of companies

from the Electronic Design Automation (EDA, [13]) and electronics industries. SystemC consists of a set of C++ class libraries plus a simulation kernel that supports hardware modelling concepts at system level, behavioural level and register transfer level. It provides a robust software environment for hardware–software co-design. Nowadays, very few tools exist that act as a bridge between the system architect modelling and verifying SoC designs in C/C++ and hardware designers implementing those designs in silicon. Moreover, until now they support only a small subset of SystemC constructs and they do not provide an optimised code yet. Two examples would be the CoCentric SystemC Compiler ([14]), which synthesises hardware from SystemC source code, and the Forter Design System's Cynthesizer ([15]), which can transform the SystemC design (using the Extended SystemC Library (ESC)) into Verilog or VHDL suitable for input into a wide variety of ASIC and FPGA logic synthesis tools;

Further work in this direction is being done within the framework of the ODETTE project ([16]). The prime deliverable of the Object-oriented co-DEsign and functional Test TEchniques project is a system for object-oriented hardware–software co-design, which provides a migration path from object-oriented system specifications to efficient hardware and software implementations. The design flow that will be supported within ODETTE is based on SystemC-plus. SystemC-plus ([17]) describes a synthesisable SystemC/C++ subset, adds object-oriented constructs to this subset, adds another class library on top of SystemC and brings post-synthesis and pre-synthesis behaviour together;

- SpecC is an example of a system level design language based on C. It is defined as an extension of the ANSI-C programming language. The first version of the SpecC language ([18]) was developed in 1997 at the University of California Irvine under the supervision of Prof. Gajski. The SpecC language is a formal notation intended for the specification and design of digital embedded systems including hardware and software. Built on top of the ANSI-C programming language, the SpecC language supports concepts essential for embedded systems design. It includes behavioural and structural hierarchy, concurrency, communication, synchronisation, state transitions, exception handling and timing ([19]). A methodology for the refinement process within the system level design flow as well as the required tools have also been developed in the meantime. Recently, a new version of the language, SpecC 2.0, has been announced which enables the migration from system level to RTL level and further implementation;

- C_x is a minimum extension of the C programming language by a process statement allowing parallel processes. This language is used for the input description of the system functions in the co-design workbench COSYMA ([28]).

2.2.3.2 Domain-Specific System Level Languages

Several system level languages have been developed in the meantime with an application domain in mind. While they are broadly used, they nevertheless comprise different restrictions depending on the domain.

- SDL: The Specification and Description Language (SDL) is an object-oriented, formal language used for the specification of complex, event driven and real time applications ([20]). The language is able to describe the structure, behaviour, and data of real time and distributed communicating systems. The great strength of SDL lies in describing large real time systems. SDL is a design and implementation language dedicated to advanced technical systems (i.e., real time systems, distributed systems, and generic event driven systems where parallel activities and communication are involved). Typical application areas are high and low-level telecom systems, aerospace systems, and distributed or highly complex mission critical systems;

- Esterel: This is both a programming language, dedicated to programming reactive systems, and a compiler which translates Esterel programs into finite state machines. It is a member of a family of synchronous languages, which is particularly well suited to programming reactive systems, including real time systems and control automata ([21]). It can generate C code to be embedded as a reactive kernel in a larger program that handles the interface and data manipulations. It can also generate hardware in the form of netlists of gates, which can then be embedded into a larger system. Esterel has been chosen by the Polis group as one of their input languages for their hardware–software co-design system;

- UML: The Unified Modelling Language ([22]) helps to specify, visualise and document models of software systems, including their structure and design. There are many UML tools available on the market for analysing the application's requirements and designing a solution that meets them. Any type of application running on any type and combination of hardware, operating system, programming language and network can be modelled in UML.

2.2.3.3 Extensions to Hardware Design Languages

Hardware description languages, for instance VHDL and Verilog, are known and used worldwide but are restricted to describing hardware. Besides design, today's language requirements comprise verification, system and software interfaces. Therefore, the classical hardware design languages might evolve to cover these other facets. One

attempt in this direction is SystemVerilog ([23]). SystemVerilog 3.0 evolves Verilog rather than substituting it. These additions extend Verilog into the system space and the verification space. It was built on top of the work of the IEEE Verilog 2001 committee. SystemVerilog improves the productivity, readability and reusability of Verilog-based code.

2.2.4 Computer-Aided Co-Design Methods and Tools

When a design team conceives a digital system, it has to perform an implementation that satisfies the system level specification within a short period of time. Furthermore, it has to maximise the system value while reducing the cost. The value of an implementation depends, for instance, on the performance and the power consumption. The cost is a function of the following parameters: number of hardware parts; size of the silicon dies (packaging costs); and software development. The value and cost are also related to the ease of debugging, testing and extending the system, as well as to the reliability and maintenance ([24]).

Computer-aided co-design tools require as input a formal system level specification in the form of system level languages or charts. It will lead to a structured design methodology, will facilitate hardware and software reuse and will permit the support of analysis and validation tools. All of these factors contribute to decreasing the design time. Furthermore, a CAD tool should be able to support the exploration of different design alternatives rapidly and should automatically generate a detailed low-level description of the implementation.

The overall objective in co-design tool research and development is providing integrated environments for concurrent specification, validation and synthesis of both hardware and software. A description of these three phases is provided below.

- Modelling. Nowadays different modelling styles are applied, each trying to fit best a specific application. In the area of modelling styles, it is necessary to distinguish between models of computation, which have an underlying mathematical structure, and hardware–software languages, which are either a means of expressing mathematical models or which describe systems and/or computation using formal semantics.

  The functional modelling of digital systems is often done using programming languages (e.g. C/C++). Its purpose is to check generic properties of the system and to derive some measures of its performance and cost.

- Validation. This phase ensures that the design is free from errors and therefore ready for implementation. As long as the system complexity increases, the

validation becomes more important and, at the same time, more difficult.

There are three main different ways of carrying out this stage: formal verification; simulation; and emulation. The simulation (or co-simulation) is the traditional way of validating hardware correctness. A simulator should provide adequate timing accuracy, fast execution and visibility of the internal registers. The existing strategies trade-off among these factors.

- Synthesis. This stage derives a detailed representation of the system implementation, starting from the system level specification. It is also called co-synthesis because it comprises both hardware and software synthesis.

  First of all, the overall system is modelled consistently and, later on, the original specification is partitioned, either manually or automatically, into a hardware and a software component. The hardware component can be implemented in terms of application-specific circuits using existing hardware synthesis tools. The software component can be generated automatically in order to implement the function to which the processor is dedicated. Synthesis must also provide a means for interfacing and synchronising the functions implemented in the hardware and software components.

  The overall system cost and performance are affected by the system level partition of the functionalities into hardware–software components. A cost function for the partitioning phase is defined, which comprises different parameters (as, for example, performance, power consumption and size). The goal is to find the most cost effective implementation by splitting its functions between hardware and software. The quality of a partition depends on performance/cost estimations based on an abstract system representation. Developing precise fast estimators from abstract models is a difficult problem.

2.2.4.1 Classification of Co-Design Approaches.

Figure 2.3 shows some of the main co-design approaches, based on their key research issue. The problem of system specification and modelling is rarely considered. Most of the work goes towards the implementation of heterogeneous systems which comprise hardware–software partitioning and co-synthesis. The complexity of the supported target architecture is one main criterion to distinguish the existing approaches.

All described approaches can be compared by their specification and implementation power ([25]). Concerning the specification, the different co-design tools are classified under their modelling style (homogeneous/heterogeneous), their features for validation (simulation or verification) and their application domain (control flow, data flow or

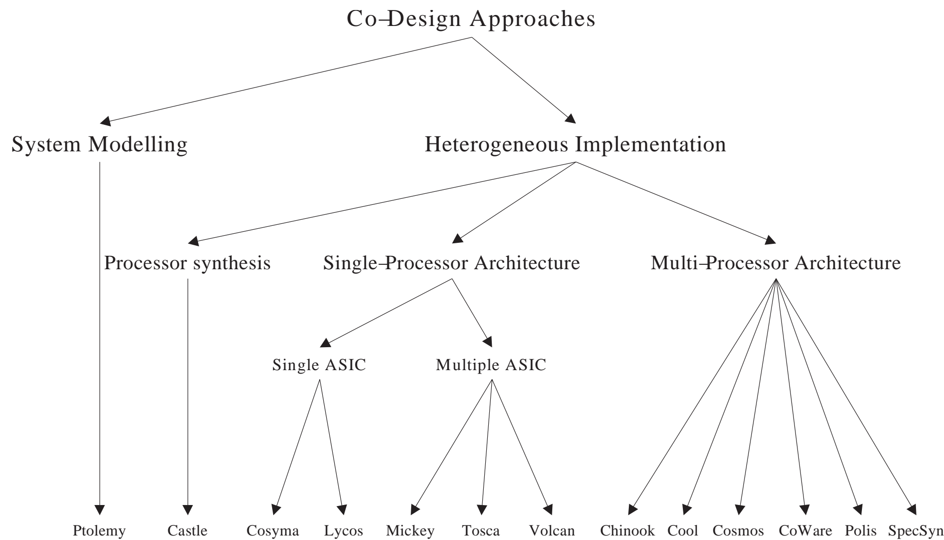

Figure 2.3: Classification of Co-Design Approaches

both). Most of the approaches offer a simulation possibility, but only POLIS supports co-verification. Furthermore, most of the approaches use homogeneous system specification and only three of them support a heterogeneous one.

In the following, a brief summary of several tools, classified according to the structure depicted in Figure 2.3, is presented.

Taking into account the implementation power, the approaches are classified by the complexity of the supported target architecture, the degree of automating the hardware–software partitioning and the support of implementing interfaces between processors and ASICs. The target architectures are further classified into the following classes: single–processor–single–ASIC architectures with exclusive or concurrent execution; single–processor–multiple–ASIC architectures; and multiple–processor–multiple–ASIC architectures. The support of implementing interfaces is divided into three classes, low, medium and high, corresponding to the degree of integrated techniques for interface and/or communication synthesis.

In the following, a brief summary of several tools studied is presented, classified according to the structure depicted in Figure 2.3:

1. Co-design for system specification and modelling;

   - PTOLEMY ([26]) is an environment for simulating and prototyping of heterogeneous systems. It allows interaction and experimentation with various models of computation, heterogeneous designs, domain-specific tools, co-

simulation and co-synthesis. The emphasis is on simulation and interaction of user-defined systems. To manage complexity, a hierarchical description is supported.

2. Co-design for heterogeneous implementation;

   - Processor synthesis;
     - CASTLE ([27]): A co-design workbench for assisting the designer to find a cost effective implementation of a system. The goal of CASTLE is to synthesise a processor and a program for this processor implementing the system behaviour. The input specification is written in a common specification language, e.g. C++, VHDL or Verilog, which is later compiled into the SIR^2 format based on control data flow graphs. Analysis tools supply the design with static or dynamic profiling information of the specified algorithm. Based on this information, the designer specifies the main structure of the processor using the schematic entry facility of CASTLE.
   - Single–processor architectures with one ASIC;
     - COSYMA ([28]): The COSYnthesis of eMbedded microArchitectures workbench was developed at IDA (Institute of Computer and Communication Network Engineering; Technical University of Braunschweig), Germany, as a platform for hardware–software co-synthesis. The resulting target architecture is limited to a processor–co-processor configuration. It covers the entire design flow including specification, partitioning and the final hardware (netlist) and software (object code) synthesis. However, there is no support for formal verification. The system is specified in C_x which is an extension of C supporting parallel processes and timing constraints. If the C_x description contains parallel processes, then they are to be scheduled and considered as a single thread of execution. The partitioning is done on basic blocks which are a sequence of statements in C_x. The partitioning process starts from a software solution and moves basic blocks from software to hardware until the timing constraints are met. The hardware synthesis is done using BSS (Braunschweig Synthesis System) which is a high-level synthesis system developed specifically for fast co-processor designs.
   - Single–processor architectures with many ASICs;
     - VULCAN ([29]): A hardware–software co-design tool similar to COSYMA, also focused on co-synthesis. The target architecture is made up of one processor and one or more ASICs, communicating through the same

communication channel and memory. The input consists of a system description in the HardwareC language and design constraints including timing and resource constraints. Then, an internal flow graph representation is performed, which is used later for performance estimation of hardware and software solutions. The automatic partitioning approach works iteratively, starting with a complete hardware solution. In contrast to COSYMA, VULCAN is able to handle multiple processes as hardware and software running in parallel.

- Multi-processor architectures.
  - POLIS ([30]): Developed at UC Berkeley, it is a framework for hardware–software co-design of reactive embedded systems. It is centred around a single finite state machine-like representation called Co-design Finite State Machine (CFSM). The system is specified using a high-level language (e.g. Esterel, graphical FSMs, subsets of Verilog of VHDL) and later this specification is translated into a network of CFSMs. The framework relies on PTOLEMY ([68]) for co-simulation and it is also easy to link it to formal verification tools. The partitioning takes place manually and requires some a-priori design experience. The paths to formal verification and simulation also provide the designer with a lot of feedback to refine the partitioning. Each hardware partition is implemented as a fully synchronous circuit where each software partition is implemented as a stand-alone C program. POLIS generates an operating system which provides communication between SW–HW and SW–SW modules, schedules the SW–CFSMs, generates device drivers for HW–SW communication and generates an event driven layer which implements the CFSM event emission/detection primitives;
  - CHINOOK ([31]): Developed at the University of Washington, this is a tool for development of reactive embedded systems. It is targeted towards interface synthesis and the co-simulation of the design before, during and after synthesis, at different levels of detail. The system specification is written in Verilog which contains both the reactive behaviour of the system as well as of the resources that will be used. The partitioning is done manually. Furthermore, it uses a static, non-preemptive scheduling algorithm. Depending on the resources and communication requirements, CHINOOK synthesises both, I/O port-based and shared memory-based interfaces. The libraries supplied to CHINOOK capture the information about the interfaces of the processors and the devices. It also provides the support for inter processor communication by synthesising the hardware and software needed to transfer data between processors.

Three of the most complete co-design approaches developed at different universities (POLIS, COSYMA and CHINOOK) have been selected and a comparison concerning their main characteristics has been performed. The results are contained in Tables 2.1.

Table 2.1: Comparison of Hardware–Software Co-Design Approaches

Feature	**POLIS**	**COSYMA**	**CHINOOK**
System spec. Language	Esterel, graphical FSM, subset VHDL/Verilog	C_x	Verilog
Constraint specification	not supported	supported	supported
Model of computation	CFSMs	RAM model	-
Concurrency	concurrent modules	single thread of execution	concurrent modules
Partitioning	manual	automated	manual
Granularity level	user-defined modules	user-defined modules	model and task level
Formal verification	supported	not supported	not supported
Co-simulation	Ptolemy	CoSim	Pia
Process scheduling	part of OS synthesis	static	static
Software perf. estimation	S-Graphs and empirical results	Sparc simulator	-
Hardware estimation	single cycle execution	List Scheduling	-
HW/SW communication	I/O ports	shared memory	I/O ports
Target architecture	processor, CFSMs	processor, coproc. & shared mem	multi proc., ASIC
Multiprocessor support	-	-	supported

Besides these academic approaches, Cadence Design Systems ([32]) announced the formation of a technology initiative, named Felix, on December 1, 1997, to deliver new methodologies and tools for virtual component-based system design. The Felix initiative pushed the work started by the Alta Business Unit of Cadence to create the industry's first development environment for rapid evaluation of architectural alterna-

tives enabled by the reuse and co-design of software and hardware intellectual property blocks at system level. The initiative sought to define a complete design methodology from the system level of abstraction to implementation. It also aimed to develop an environment consisting of design entry mechanisms, design languages, synthesis and analysis tools including simulation, various forms of formal verification, and hardware and software synthesis. Further, it sought to create a set of modelling principles to guide the development of models at the appropriate levels of abstraction, with precise accuracy and execution speed trade-offs. Many concepts in the Felix initiative were pioneered by the POLIS research project. The outcome of this initiative is the industry's first system level development environment for platform-based hardware–software co-design for IP reuse, the Cadence Virtual Component Co-design (VCC) ([33]). The Cadence VCC environment clearly differentiates between a behaviour model, which identifies what the system does, and an architecture model, which identifies how the system is implemented. This differentiation allows system designers to simulate the performance effects of a behaviour running on a number of different architectures early in the design cycle.

Smaller approaches, such as Visual Elite from Summit ([34]), offers a single environment for system level designing, down to the RTL and gate level. The architectural and performance analysis is performed by the System Architect tool, the co-verification of hardware and software blocks utilising Virtual–CPU and the hardware–software interface design with Regent.

Both development environments, i.e. VCC and Visual Elite, are prepared for testing different hardware–software architecture–partition alternatives. Nevertheless, it becomes a time consuming task due to the extensive customisation required and the necessary rebuilding of the structure in case new building blocks have to be added. The high modelling effort and the tool-specific description language are two additional aspects that should be improved by the existing system level development environments.

2.3 Performance Estimation Approaches

The increase in the integration and complexity of the systems has promoted the development of fast and accurate analysis techniques for various metrics such as performance, power, system cost, etc., for guiding the partitioning–mapping step. These techniques differ, first of all, in the target metric and, secondly, in the level of abstraction and method applied. The present work deals with the metric performance and therefore an exploration of the existing techniques in this area is presented further on.

When estimating of the performance reached by the system under study, two steps have to be carried out. The first one, the so-called block level performance estimation, considers the performance of each individual functional block within the system target architecture. In the second one, the system level performance estimation, carries out an estimation of the whole system, including the communication structure. In the first case, a further distinction is made between the techniques dealing with the estimation of hardware and software implementations, respectively.

Performance analysis and estimation methods can broadly be divided into static and dynamic techniques ([35]). Static techniques are concerned with the analysis of a specification without simulating it. In most cases, it is used for worst case analysis and is suitable for finding cases that are hard to cover with simulation. On the other hand, dynamic techniques are concerned with the analysis of runtime behaviour, and rely on test vectors and input scenarios. They are preferred when the system performance is heavily dependent on the input data, and in cases where the average performance is more relevant than the worst case behaviour.

2.3.1 Block Level Performance Estimation

The analysis techniques with regard to each building block can be divided in those dealing with the runtime inside a processor (the so-called software performance estimation techniques), and those dealing with the hardware units (the so-called hardware performance estimation techniques).

2.3.1.1 Software Performance Estimation Techniques

The choice of the algorithm to be implemented has a large impact on the performance of embedded real time systems. Therefore, performance estimation of embedded software is vital in an early design phase. The performance of software in embedded hardware–software systems depends on the structure of the software program as well as on the

components of the target system. An effective estimation procedure has to model both the target system and the software program at an abstraction level that makes the estimation time reasonable without losing too much accuracy ([36]).

The level of abstraction characterises different software performance estimation techniques. They are classified into four groups by [37]:

2.3.1.1.1 Cycle-Accurate Performance Model. Definition and implementation of a cycle-accurate Instruction Set Simulator (ISS) which is used to run the code. This model can consider and analyse the software behaviour. The approach is precise but slow and requires a detailed model of the hardware and software. The performance analysis takes place after completing the design, when architectural choices are difficult to change. Besides, an ISS can seldom be reused due to the fact that it has been built for a special situation.

An approach to integrate a clock cycle-accurate ISS with a fast event-based system simulator is proposed in [38], which claims that other approaches are not accurate enough for hard real time systems and complicated designs. The proposed scheme is especially effective for applications where the delay of basic blocks is approximately data independent.

2.3.1.1.2 Timing Annotation of the Control Flow Graph (CFG). A Control Flow Graph is built on the basis of the compiled software description. It is further annotated with the required information to derive a cycle-accurate performance model (e.g., with regard to pipelining and cache). The graph is made up of edges and basic blocks. Each basic block corresponds to a function block of a basic function in the assembly program. The compiled code for each basic block is analysed and by means of the received information, each basic block is annotated with its weight. The nodes in the graph are allocated depending on their weight. The optimisations of the compiler and the current selection of commands are included during the analysis.

In [39], a compilation-based software performance analysis method is presented. It combines a state-of-the-art optimising compiler with a high-level co-simulation and co-design methodology. The key idea is to use the GNU-C compiler to generate assembler level C code. This code is annotated with timing information and used as a very precise and fast software simulation model.

2.3.1.1.3 Original C-Code Annotation. Annotation of the original C-code with timing estimates trying to guess compiler optimisations. It has the advantage of not requiring a complete design environment for the chosen processor, since an estimation

of the execution time for each high-level language statement on the chosen processor is predefined. However, it cannot consider compiler and complex architectural features.

In [36], two estimation methods at different levels of abstraction (s-graph (software graph) level and CFSM (Codesign Finite State Machine) level) for use in the POLIS hardware–software co-design system ([67]) are presented. The s-graph level method, which concerns the software branch, reaches an accuracy of 20% when compared to an assembly level analysis. The work introduced in [35] proposes a high-level estimation technique that both estimates the performance and computes the expected accuracy. The accuracy is then used to provide a confidence interval to the estimated performance.

2.3.1.1.4 Software Estimation by Means of Linear Equations. Use of linear equations to implicitly describe the feasible program paths. It has the advantage of not requiring a simulation of the program, hence it can provide conservative worst case execution time information. Static estimation is insensitive to input data. It just computes the average number of clock cycles needed to execute the program. Static estimation can yield good results if the number of loop iterations is known and the conditional branching probability can be predicted correctly.

The method presented in [40] claims that static estimation has a number of advantages since it takes much less time and space than dynamic simulation and it does not need input data. The proposed method uses a probability-based flow analysis technique and applies it to the performance estimation for system level specifications. Three software metrics are then provided: execution time; program memory size; and data memory size for a specification executing on a given processor. A study of the different performance metrics that need to be considered in this context is shown in [41]. Furthermore, it examines a range of techniques that have been proposed for static timing analysis, which can broadly be classified into path analysis and system utilisation analysis techniques. They are interdependent, and thus need to be considered together in any analysis framework.

2.3.1.2 Hardware Performance Estimation Techniques

The estimation of the hardware runtime cannot be considered separately but has to be done in connection with a hardware effort estimation.

Concerning the hardware effort, the estimation technique introduced in [42] adds together the hardware effort for modules, registers, multiplexers and the control unit. However, the last unit, the controller of a hardware design, is very specific to the applied synthesis tool. In [44], a method to rapidly estimate hardware size during the functional

partitioning is introduced. The method includes a data structure which represents a design model, and an algorithm that incrementally updates the data structure during functional partitioning.

Once the hardware resources are fixed, the hardware runtime can be estimated. The requirements for developing an adequate algorithm that estimates the hardware runtime are: high precision; low computational effort; independency of further design steps; and easy adaptation to the following synthesis tools. The first requirement excludes all estimation approaches that are limited to the scope of basic blocks because they are not able to identify the global optimisation potential in hardware synthesis.

The hardware runtime estimation within the scope of hardware–software partitioning should be able to achieve the scheduling's results of the following design steps with a high degree of accuracy. There are some fast but not sufficiently accurate approaches based on lower bound methods which do not assume that a scheduling has already been performed, e.g. [43]. On the other hand, the technique presented in [45] performs a path-based scheduling before the estimation is started. The estimation takes place at a Control and Data Flow Graph (CDFG) representation that is directly derived from a C system level description. The scheduling methods that can be applied are mainly divided into two groups, as cited in [46]:

2.3.1.2.1 Local Scheduling Methods A separate scheduling for each data block is performed, and therefore it is restricted to the borders of the respective block. Thereby, these methods can neither go into the parallelism of the system nor reduce the complexity of the complete application. Through this, the results delivered by these methods are not very reliable and precise.

2.3.1.2.2 Global Scheduling Methods An optimisation over the border of basic blocks takes place. It achieves a good precision by scheduling of parallel processes. It is adequate for control-dominated systems, where there are more control than arithmetic operations. There are three well-known global scheduling methods, which are still in a research phase. They are: Percolation Scheduling; Speculative Execution; and Path-Based Scheduling.

- Percolation Scheduling: In this method there exist four main base transformations: *move-up*; *move-cj*; *delete*; and *unify*. With the help of such transformations, the program is reorganised in order to achieve a higher parallelism;
- Speculative Execution: In contrast to the accepted methods, the Speculative Execution utilises the control dependencies of the application to obtain an enhancement of the speed. Concretely, the control dependencies are ignored, i.e.,

assignments are executed at a point in time, when it is not already fixed whether the program execution will achieve the program branch with such assignment. Speculative Execution is then reasonable when a further parallelism of the application is limited by the program structure;

- Path-Based Scheduling: This scheduling method takes each possible program flow path and performs an ASAP (As Soon As Possible) scheduling on it. Afterwards, all of the paths are superposed to achieve an entire scheduling. From each conditional jump instruction at least, two paths arise. Here, the problem of path explosion will be noticeably fast. In the worst case, the paths increase exponentially with the number of conditional jump instructions. To avoid this problem, different possibilities can be accessed, as the one presented in [46], where intersections in the graph are inserted additionally. This technique allows a substantial reduction of the possible paths, nevertheless a list of criteria when choosing the intersections has to be kept in order not to degrade the results.

2.3.2 System Level Performance Estimation

The system level performance estimation techniques perform an estimation of the whole system, including the communication structure. The existing methods for automatic partitioning either ignore inter-component communication entirely or use simple models of communication to guide the partitioning–mapping step. Refining the abstract communication of the system into a specific communication architecture (with associated communication protocols) is performed as a subsequent step in the system design flow. In practice, while these two steps of system design are sometimes treated as separate problems for reasons of tractability, the importance of integrating communication protocol selection with hardware–software partitioning is clearly demonstrated in [47]. Furthermore, [55] shows that the synchronisation overhead associated with inter-process communication can contribute significantly to the overall system performance.

The work dealing with the system performance analysis to help in the design of high performance communication architectures is very small in comparison with the work focusing on the partitioning–mapping step. The techniques that do consider the effects of the communication architecture can broadly be divided into the following categories:

2.3.2.1 Simulation-Based Approaches

These techniques are based on an evaluation of the system model by means of simulation. For this purpose, different levels of abstraction in the modelling of the com-

ponents and their communication are used. The level of abstraction for modelling the communication allows for a trade-off between simulation time and accuracy.

The approach presented in [48] introduces a hardware–software co-simulator which provides substantial speed-ups over traditional co-simulation methods by permitting dynamic changes in the level of detail when simulating communication channels between system components. As part of this simulation environment, implemented as a PTOLEMY domain ([50]), [49] introduces a new language for component and interface specification, the Pia language. The importance of a well-defined modelling language to support the development an executable model that properly represents the system to be designed is also addressed in [51]. The Parallel Object-Oriented Specification Language (POOSL) is introduced here and the simulation method towards the performance estimation is explained.

However, these techniques still require a simulation of the complete system, limiting their computational efficiency. In order to substantially improve the performance without sacrificing user access to detail, [52] proposes a simulator which separates communication from behaviour.

Probabilistic models have also been used in simulation-based approaches, as the method presented in [53] has done. This work describes a faster technique as compared to cycle-accurate simulation. The methodology is based on probabilistic modelling of system components customised with application behaviour.

2.3.2.2 Trace-Based Performance Analysis Strategies

These are generally based on trace transformation techniques. A trace is a track of the system execution that contains information concerning computations and communications of the system components. These methods take as input a trace generated by an application process and generate, as output, a trace accepted by an architecture model and which contains the architecture level operations. Thus, a trace transformation provides the mapping of application level communication primitives onto architecture level communication primitives.

This basic idea of collecting an execution trace and using it for performance estimation has been used in the field of high performance processor design, for instance for cache simulation ([59] and [60]).

The work introduced in [61] focuses on the mapping and exploration stage in the design of embedded signal processing systems. In particular, it covers the mapping of primitives used for expressing communication behaviour at the application level onto primitives used to implement the communication in architectures. It uses the per-

formance analysis tool SPADE ([62]) (System level Performance Analysis and Design space Exploration) to provide accurate feedback on the performance of application–architecture–mapping combinations.

In [63], a new simulation method is performed that is also based on the disjunction between both models — the algorithm and the architecture. This work illustrates that by means of a joint simulation of complementary models it is possible to perform a simulation of the architecture for data dependent algorithms. It also demonstrates that the comparison of different architecture alternatives can easily be carried out by the use of interchangeable architecture models.

Furthermore, there are some hybrid trace-based performance analysis methodologies, as for example the one presented in [64] for driving the design of bus-based SoC communication architectures, and in [65] and [66] for multi-channel communication architectures. An initial co-simulation of the system is performed, with the communication described in an abstract manner (using POLIS [67] as a hardware–software co-design tool and PTOLEMY [68] for system level simulation). An abstract set of traces is then extracted. It contains the information about the computations and communications of the system components. These traces are represented as a Communication Analysis Graph (CAG), whose analysis provides an estimation of the system performance as well as various statistics about the components and their communication. An important feature of this approach is its ability to model various dynamic effects of the communication architecture and to consider the inter-dependencies between the computations, synchronisations, and data communications performed by the various components while estimating the system performance.

2.3.2.3 Static Performance Estimation Methods

These techniques try to avoid the computationally prohibitive alternative of exhaustive simulation, arguing that an efficient exploration of the system design space necessitates fast performance estimation. They include models of the communication time between components of the system and often assume systems where the computations and communications can be statically scheduled ([54]). The communication time estimations used in these systems are either overly optimistic, since they ignore dynamic effects such as wait time due to bus contention, or are overly pessimistic by assuming a worst case scenario for bus contention. This last case is addressed, for instance, in [55], which issues a worst case performance analysis of a system described as a set of concurrent communicating processes.

Several techniques use different kinds of graph models suitable for hardware–software partitioning of single processes. For example, [56] presents a Control Data Flow Graph

(CDFG) and the required transformations on it before partitioning in order to achieve a structure that makes an accurate estimation of the communication overhead between nodes mapped to different processors feasible. Another interesting approach pursuing the performance estimation for hard real time systems is presented in [57] and [58]. It is based on an abstract graph representation, a Conditional Process Graph (CPG), which captures at process level both the data and control flow. The analysis of the graph allows the reduction of the pessimism of the estimation by using the knowledge about the data and control dependencies.

Finally, [76] presents a methodology for performance prediction of parallel programs that comprises a procedural application and machine specification paradigm based on the performance simulation language PAMELA ([77]). The analytical evaluation is based on conventional static program analysis techniques called serialisation analysis. This evaluation technique is only able to handle simple models with fixed delays.

2.3.2.4 Analytical Performance Estimation Methods

The analytical methods are based on models of the tasks related to the application domain and a means of characterising the performance of the target architecture. The goal of such methods is to quickly identify interesting architectures, which may then be subjected to a more detailed evaluation.

In [69] and [70], a novel analytical method to explore the design space of packet processing architectures on system level is presented. The proposed framework consists of a task ([71]) and a resource model and a real time calculus which is used to analytically determine properties such as delay and throughput. The design space exploration is posed as a multi-objective optimisation problem. To speed-up the exploration, it uses several linear approximations in the real time calculus ([72]).

The approach proposed in [73] extends the POOSL language ([74]) with the capabilities to express probabilistic behaviour and to analyse performance figures analytically. The key idea is the development of a real time probabilistic process calculus that is able to express the basic concepts of POOSL. Furthermore, [75] describes a performance analysis algorithm for a set of tasks executing on a heterogeneous distributed system. This algorithm handles both upper and lower bounds on process execution time, considers data dependencies between processes and handles preemption and task pipelining.

Completing the system performance estimation methodologies, [79] proposes an approach which takes advantages of both system and RT levels of abstraction and com-

bines both static and dynamic analysis techniques in order to obtain the best trade-off between speed and accuracy. The four main steps in the overall flow are: execution time computation of basic elements; back annotation; architecture modelling; and system level simulation.

A comparison of the above-mentioned approaches for system performance estimation can be found in Chapter 3. On the basis of this comparison, the advantages and disadvantages of the previous methods are extracted and the need for a new strategy is founded.

2.4 VLSI Architectures for Networking Applications

The main application scenario of the current work, VLSI networking architectures, has undergone continuous evolution as the speed and functionality of local and wide area networks have grown. Both performance and flexibility requirements have to be fulfilled by the VLSI networking devices. Among the performance requirements, the increasing number of users, the extensive applications (video, audio) and the LAN (Gigabit Ethernet) and WAN speed (OC-48, OC-192) can be mentioned. The most suitable design solution to reach these demands is their implementation as hard-wired logic (ASIC). On the other hand, it has to be considered that a rising flexibility to cope with the new data protocols (e.g. IP v.6), to support advanced services (e.g. Quality of Service) and to reduce the time-to-market is essential. And this requirement of flexibility can only be reached by using programmable hardware, i.e., FPGAs or processor cores. To meet the functionality and performance requirements of present and emerging network applications, the current trend is to use Programmable microprocessors on Network Interfaces (PNI) that can be customised with domain-specific software ([80]). This trend has created the so-called Network Processors market niche. They deliver hardware level performance to software programmable systems. This powerful combination offers a revolutionary approach to the design of communication systems.

In order to understand the new aspects and characteristics offered by Network Processors, an overview of the network infrastructure as well as the evolution of the VLSI networking architectures design is presented below.

2.4.1 Network Infrastructure

Nowadays, data networks carry a broad variety of traffic and therefore a broad range of infrastructure equipment becomes necessary. The decision as to which equipment is required in a particular segment of the market can be made by considering four main factors: speed; cost; intelligence; and flexibility ([81]). Thus, a trade-off among them has to made.

Before pointing out the main market segments of network equipment, an introduction to the layers and functionalities involved in a computer network is necessary.

2.4.1.1 Computer Networks

Internet, can be defined as a method of interconnecting physical networks and a set of conventions for using networks that allow the computers they reach to interact ([82]).

During the early stages of building the Internet the main goal was to set up a decentrally organised interconnection of computers with redundancy so that a breakdown of a part of the network would not affect the connectivity and efficiency of the overall Internet. Consequently, every computer attached to a network is assigned a unique address. Since routes through the network are not determined statically but dynamically depending on the current state of the network, every single packet must be processed by every intermediate network node from the source to the destination of a transmission. Packets from different transmissions should be treated equally by the network and nodes make a best effort to handle all packets in the order of their arrival.

2.4.1.1.1 Communication Layers in a TCP/IP Network. The communication over a data network is influenced by different problems such as hardware failure, network congestion, packet delay or loss, data corruption and data duplication or sequence errors. One solution for coping with these problems is the use of complex data communication systems that do not rely on only a single protocol for handling all transmission tasks. They require a protocol family or protocol suite.

Hosts in the Internet can be reached through routers. The routers decide the path that packets should follow. The Internet Protocol Suite (TCP/IP) provides the rules for communication, i.e., among others it defines the details of message formats, determines how the networking equipment responds when a message arrives, defines how a computer handles errors or other abnormal conditions and provides computer communication independency of any particular vendor's network hardware. The TCP/IP software is organised into five conceptual layers. Figure 2.4 shows these layers as well as the type of data as it passes through a router.

Each layer can only pass information to the next higher or lower layer through defined interfaces. At each layer, protocols define the operations and responses necessary to exchange information between peer layers at different network nodes. This information is held by layer-specific header fields that are added to traffic entities. Lower layers only consider the transmission of traffic between neighbouring network nodes whereas the higher layers affect the end-to-end transmission through several intermediate nodes. The Open Systems Interconnection (OSI) reference model by ISO is composed of seven abstract layers. Nevertheless, as already mentioned above, the Transmission Control/Internet Protocol (TCP/IP) stack used by the Internet only considers five of them ([83]). They are:

1. Physical Network: This is the lowest layer. It considers the plain transmission of data streams through a physical medium, e.g., a copper wire, between neighbouring nodes;

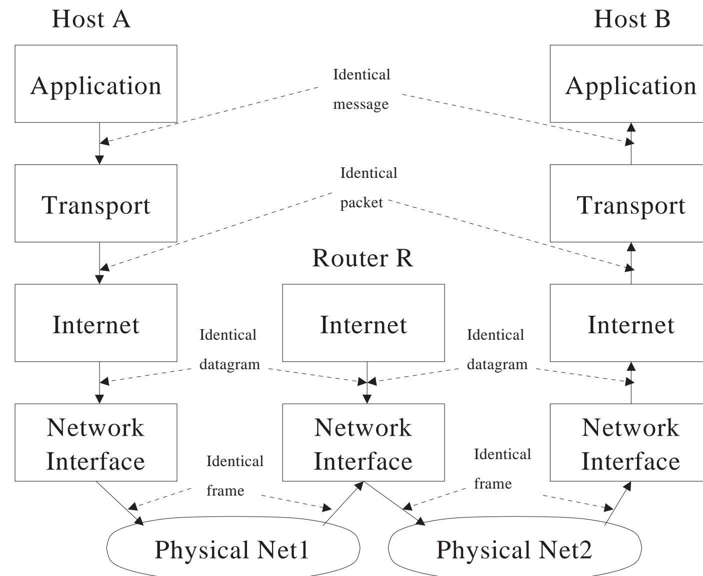

Figure 2.4: TCP/IP Conceptual Layers

2. Network Interface: This layer is responsible for reliable transmission of data units (frames) between neighbouring nodes;

3. Internet layer: This is the lowest layer that affects the plain end-to-end transmission of data packets. It defines the basic unit of transfer across the network and includes the concepts of destination addressing and routing;

4. Transport layer: The transport layer is responsible for end-to-end transmission of aggregated packets, the so-called segments or messages. A reliable transmission may be enabled by packet sequencing and flow control;

5. Application layer: This layer deals with the exchange of data between applications running at different network nodes. For instance, the FTP protocol handles whole file transfers and the HTTP protocol is responsible for web page downloads.

The TCP/IP protocols are extremely flexible in that almost any underlying technology (e.g., Ethernet, FDDI, ATM, ARPANET and X25) can be used to transfer TCP/IP traffic.

2.4.1.1.2 Communication Processing Tasks. There are two broad categories of communication tasks as can be seen in Figure 2.5 ([84]):

- Forwarding Plane tasks: They perform operations on the forwarding path, where the data communication occurs in real time. These constitute the core operations for a networking device, and hence are performance critical. In a switch or a router, these are the functions that receive, process, and transmit packets into and out of the device;
- Control Plane tasks: They perform less time-critical control and management functions that determine general device operation. In a switch or a router, these functions control routing table maintenance, port states, and higher level management.

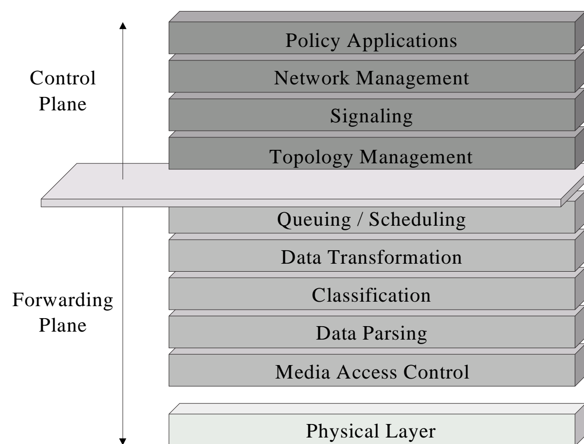

Figure 2.5: Communication Processing Tasks

From now on, more attention is given to the forwarding plane tasks, where a high speed path is required. These forwarding functions include some common functions encountered in every router or switch and other functions related to advanced services within the Internet, which are not yet included in every router or switch. The first three functions of the following list belong to the first group, whereas the last two belong to the second group.

- Media Access Control: Low layer protocols, such as Ethernet, SONET framing, ATM cell processing, etc., are implemented. These protocols define how the data is represented on the communication channel and the rules governing how that channel is accessed. There has been a great standardisation among network devices in this area (due to standard-based protocol definitions) and, at the same time, is an area of greatest diversity (due to the wide and ever-growing variety of protocols);

- Data Parsing: Cell or packet headers containing relevant addresses, protocol information, etc., are parsed. Switching devices today need the flexibility to gain access to and examine a wide variety of information at all layers of the OSI model. And they have to perform it in real time and on a conditional packet-by-packet basis;

- Data Transformation: Data within or between protocols is modified or translated into other formats. It can range from address translation within a given protocol (such as IP) to full protocol encapsulation or conversion (such as between IP and ATM);

- Classification: A packet or cell is identified against a set of criteria defined at layers 2, 3, 4 or higher of the OSI model. Once data is parsed, it has to be classified in order to determine the required action. These actions might include such basic functions as a filtering/forwarding decision, as well as advanced QoS and accounting functions based on a specific end-to-end traffic flow;

- Traffic Management: This group comprises forwarding and control plane tasks. The forwarding functions include the queueing, policing and scheduling of data traffic through the device according to defined QoS parameters, based on the results of classification and established policies. These functions are key to supporting convergence of voice, video and data in next generation networks.

2.4.1.1.3 Classification of Computer Networks. Communication networks can be classified under different criteria ([86]), according to geographic coverage and according to connectivity. Under the first criterion, the Local Area Networks (LAN) interconnect end devices within a relatively small area, while the Wide Area Networks(WAN) interconnect LANs. A WAN network of intermediate extension limited to a town or a city is also called a Metropolitan Area Network (MAN). Under the second criterion, the connectivity, it can be differentiated into Autonomous Systems (AS), the access network (where an Internet Service Provider (ISP) gives the Autonomous System access to Internet) and the core or backbone network. Nevertheless, an AS can also be composed of an access/core network. An additional level can be defined between the access and the core network, called the distribution network. The distribution network provides the transit between the access and core parts of the network.

2.4.1.2 Network Equipment

Traditional switch and router definitions are no longer applicable as capabilities have expanded and overlapped. It is now more useful to look at where the equipment resides

in the network than at what its internal architecture is ([86]). Concerning this criterion, a distinction can be made between service provider and access equipment.

- Enterprise Equipment. In the enterprise environment, Ethernet LAN switches make up the bulk of the infrastructure. At the workgroup level, layer 2 switches provide low cost and high port densities. Higher in the network hierarchy, backbone switches aggregate the bandwidth from many workgroup switches. Backbone switches add Layer 3 and higher capabilities, replacing the traditional role of routers in the LAN backbone. Routers have been relegated to the edge, making the connection between the LAN and the WAN;
- Service Provider Equipment. At the edge of the WAN we find edge routers, which must handle many protocols. These devices handle lower speed links but must have a great deal of intelligence and flexibility. Behind these devices is the optical core of the WAN. Here, switching must occur at wire speed over high speed links. ATM switches comprise most of todays WAN core;
- Access Equipment. It can be thought of as the glue connecting users to the service providers. Much of this equipment serves as an aggregation point.

2.4.2 VLSI Networking Architectures Design

The design of the network equipment powering the Internet revolution has brought about rapid changes over the last decade. Nowadays, the network equipment vendors are racing to provide the newly converged voice-video-data communication infrastructure. Furthermore, conservative estimates for aggregate bandwidth on the Internet backbone indicate a doubling each year for the past ten years, and further expansion at these levels is likely to continue for some time ([89]). One consequence of this growth is the demand for greater performance, flexibility, reliability and cost effectiveness in the primary components (e.g., routers and switches) which control the flow of data through the network ([87]).

2.4.2.1 Evolution of VLSI Networking Architectures

In former times, networking devices were built as a combination of general purpose CPUs, discrete logic and ASSP (Application Specific Standard Products). Their software-based nature was the key to adapting to new protocol standards and the additional functionality required by the networks. Although these designs were large, complex and slow, they met the needs of early networks.

Over time, as network interface speed and density increased, the performance of general purpose processors was no longer sufficient. This led network vendors to develop simpler, fixed-function devices that were built on the basis of ASICs. These devices traded off the programmability of software-based designs for hardware-based speed. As ASIC technology progressed, more and more functionality was incorporated into the hardware. This was in part possible thanks to the protocol consolidation around IP and Ethernet as the dominant enterprise network technology, which reduced the need for product flexibility.

Today, the situation has substantially changed. The convergence of public voice and data networks is leading to increased time-to-market pressure and shorter product life cycles. And this is happening just when product development cycles are growing due to complex ASIC designs and associated software re-designs. Furthermore, although IP is emerging as the dominant protocol, new IP capabilities, such as Quality of Service (QoS), are appearing. In addition, the number of different interface types, ranging from sub-T1 through OC-48 in the WAN space and to 10/100 and Gigabit Ethernet in the LAN area, is increasing rather than decreasing. As a result, networking products require the same programmability and flexibility that was available in the early CPU-based architectures in order to quickly adapt to emergence standards, while maintaining the performance gains achieved through ASICs. These requirements have pushed the emersion on the market of the so-called Network Processors (NP). They deliver hardware level performance to software programmable systems. In Figure 2.6, the area covered by each of the design solutions mentioned above concerning performance vs. functionality and flexibility is shown.

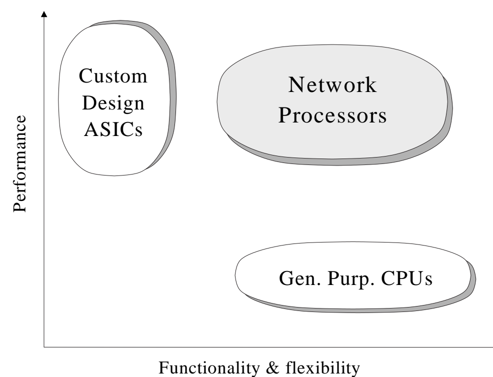

Figure 2.6: VLSI Networking Architectures

2.4.2.2 Design Trade-Offs

Switches and routers have been designed using a mix of standard RISC processors, ASICs, and ASSPs. RISC processors are used as the host processor, handling control and management functions. ASSPs are used on line cards to provide Medium Access Control (MAC), framer, and physical interface functions. Everything between the CPU and the ASSP has been the domain of ASICs. ASICs have implemented linecard interfaces to the proprietary backplane bus or switch fabric. Until recently, most switch fabrics were developed using ASICs. Moreover, protocol acceleration functions were implemented in ASICs. Network Processors are primarily intended to replace ASICs.

2.4.3 Network Processors

The previous Section depicts the evolution of VLSI networking architectures which has led to the development of the Network Processors. Many different architectures for NPs that reach high speeds and offer flexibility have appeared in the last while thanks above all to multi-threading software engines and powerful co-processors. Network Processors are suited to process packets at high speed and are adaptable to other/new complementary services.

With port speeds up to Gigabit Ethernet and OC-12, first generation network processors had only two strong market target segments: LAN backbone switches and edge routers. Second generation network processors are targeting speeds up to OC-192.

2.4.3.1 Common Characteristics

For the conception of these new architectures, the following characteristics of the application scenario, i.e., networking packet processing functionalities, have been taken into account.

Packet streams only have dependencies among packets of the same flow but none across different flows. This ensures that the processing can easily be distributed over several processors. That is, there is an inherent parallelism associated with the processing of separate independent packet flows. Thus, the problems of complex synchronisation and inter-processor communication that are typically encountered when dealing with parallelisation of many traditional computer applications is not present. Therefore, the performance of the packet processing within a router can be increased considerably by dividing the tasks and using several processing units of modest speed. Moreover, the tasks involved in the processing of a packet is fairly simple, mainly consisting of extracting data from a bit stream and doing some pattern matching or table

lookups. As a result optimised processors can be developed for this purpose (usually referred to as packet engines), which fit into only a few square millimetres of silicon.

Typically, packet engines are multi-threaded in order to take advantage of the large number of incoming packets. Thus, each engine holds one or more packets while it processes the current one. If the processing of the current packet stalls (for example, because of a memory access), the engine switches to the next selected context which holds a packet. In this way, the engine does not waste time waiting for the memory response.

Many Network Processors supplement the packet engines with co-processors, which carry out costly functions such as packet classification or policing. They are usually implemented as fixed function logic blocks. Their functions may nevertheless be configurable.

Two further characteristics of the existing Network Processor architectures are the inclusion of a general purpose processor for performing control plane tasks and the implementation of four basic external interface types: Line interface; fabric interface; memory interface (often consists of several physical connections); and interface to the host.

2.4.3.2 Main Attributes

After presenting the common characteristics of Network Processors in Section above, their key attributes can be summarised in the following points ([85]):

- Programmability: A Network Processor supports a wide range of interfaces, protocols (for packets, cells and data streams) and product types. This requires programmability at all levels of the protocol stack, from layer 2 through layer 7;
- System Flexibility: Software implementation of the functions between the physical interfaces and the switching fabric allows simpler upgrade paths. It is an important feature when looking at the constantly changing networking world;
- Processing Capabilities: An optimised processing architecture is required (up to Gigabits per second) to support wire speed operation at high bandwidths and still have processing headroom for advanced applications;
- Functional Integration: A high-level of system integration reduces part count and system complexity. At the same time, it improves the performance compared to a design that incorporates multiple components and also avoids the interconnection bottlenecks;

- Simple Programming Model: The target code for a Network Processor might be easily understandable by the developer in order to be useful. Programming in C/C++ has two main advantages: Millions of skilled programmers and many more lines of code already exist and it enhances the future portability of the code base, enabling use in future generations of Network Processors and industry standard programming interfaces;

- Open Programming Interfaces: The processor's architecture must support generic Communications Programming Interfaces (CPI) to simplify the programming task and allow future software reuse and software reliability.

The key attributes concerning speed and flexibility pursued by the Network Processors introduce in their design a scenario of multiple alternatives. The evaluation of every architecture–partition alternative for such complex systems at a low-level of detail would take a long time. Because of the time-to-market pressure, this cannot be any more affordable. Therefore, the selection of the optimal target architecture as well as the partitioning of the design functionalities that keeps the constraints should start at a high-level of abstraction, i.e., at system level, where the designer has more freedom to evaluate different trade-offs.
Moreover, from the point of view of the user of a certain Network Processor, the decision concerning the best allocation of the functionalities to be implemented onto the corresponding architecture can be facilitated if a model of the system at a high-level of abstraction is provided.

Chapter 3

Performance Estimation for Design Space Exploration

3.1 Chapter Introduction

Nowadays, the product life cycles become shorter and shorter, rendering rapid design cycles a critical issue. This creates the need for a fast exploration process of design alternatives, where the performance estimation plays a decisive role in ranking the design alternatives. If this exploration is performed at a higher level of abstraction, a huge scenario of low level alternatives is covered, as can be seen in Figure 3.1. Although no low level details can be observed and investigated at a high level of abstraction, it is sufficient to discriminate among several different alternatives and reduce the design space. Only the best alternatives will be further evaluated at lower levels of detail.

Design space exploration can be considered one of the most important issues within system level design. It tries to find the best architecture–partition alternative, which means the optimal partition of the system functionalities between hardware and software and, at the same time, the right architecture resources (hardware components and communication protocols). Performance estimation is characterised by measuring utilisation and throughput of the resources.

Addressing the design space exploration, the proposed methodology introduces a fast procedure which is based on a reconfigurable functional graph. This new methodology allows a fast evaluation of a broader range of possible architecture–partition alternatives without requiring a re-building of the initial functional graph. It is only necessary to provide the information concerning the new architecture–partition configuration to be tested.

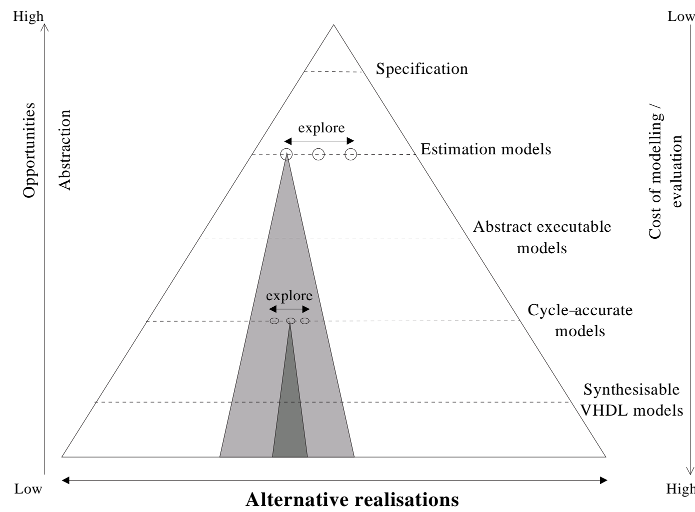

Figure 3.1: Abstraction Levels for Models

First, the basic concepts behind design space exploration and performance estimation are introduced. Next, a comparison of existing approaches is performed and, thus, the need for improvements deduced. Then, the integration of the proposed performance estimation method within the design flow is depicted. Last, the boundary conditions and the assumptions made for the development of the proposed methodology are presented.

3.2 Design Space Exploration

When facing a new application-specific design the designer tries to find a feasible solution that satisfies the constraints, such as real time deadlines, and that also optimises design objectives like cost and power. The challenge is, thus, to select the architecture such that the design fulfills the requirements. The architectural choices inherently represent a broad scenario. If choices of partitioning and mapping are also considered, the exploration becomes more and more complex.

Traditionally, hardware–software design methodologies have started from a predefined target architecture. They include hardware–software and interface synthesis along with partitioning and mapping. Such methodologies deliver accurate results for evaluation, but have drawbacks in terms of speed and cost due to the fact that the tasks involved require a lot of effort and are very time consuming. In contrast to the classical methodologies, the exploration iteration at the system level includes only the partitioning and mapping step. It is a faster scheme, but needs some techniques to

estimate the performance parameters from abstract models for each component and mapping. Moreover, partitioning and mapping are NP-complete ([90]) problems and, therefore, there is a huge space for exploration, making good heuristic algorithms and methods necessary.

The designer has to make some trade-offs in the exploration process. At the system level, a larger part of the design space can be explored in a given time. Although it is less accurate than an exploration at a lower level of abstraction, it helps to narrow down the design space. Once it becomes smaller, the exploration can include lower design steps. The exploration is then more accurate at the expense of taking longer to construct, evaluate and change the corresponding models.

Exploration is an iterative process where each iteration comprises different steps of the system design. Figure 3.2 depicts the framework followed by the proposed method for the design space exploration at the system level. At the starting point, the functionalities of the system to be implemented are described. And, at the same time, a target architecture for the implementation is selected. In an intermediate stage, the estimation of the performance achieved by every functionality from the specification mapped to the possible processing units of the target architecture is performed and stored in a library. After carrying out the mapping and scheduling of the functionalities onto the target architecture, the estimation of the system performance value for the selected architecture–partition alternative is calculated. Only when the performance constraints are met, are further steps in the design flow procedure carried out; otherwise a new mapping has to be tested. Furthermore, if no mapping meets the constraints, a re-allocation of the target architecture is required. This leads to a re-organisation and/or addition of components or communication primitives.

Next, the challenges introduced by the design space exploration of complex systems is illustrated by means of a mathematical formula. Such formula computes the number of possible alternatives for mapping a given number of functions onto different target architectures with a variable number of processing units and different kinds of technology for each processing unit.

The number of solutions for mapping a system specification made of n tasks (i.e., functions) on an architecture made of k no-empty modules (i.e., processing units) may be computed using the *Stirling numbers of the second kind* ([96]), *S(n,k)*. *S(n.k)* is given by the following sum:

$$S(n,k) = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n \tag{3.1}$$

They satisfy the following recurrence relation, which forms the basis of recursive

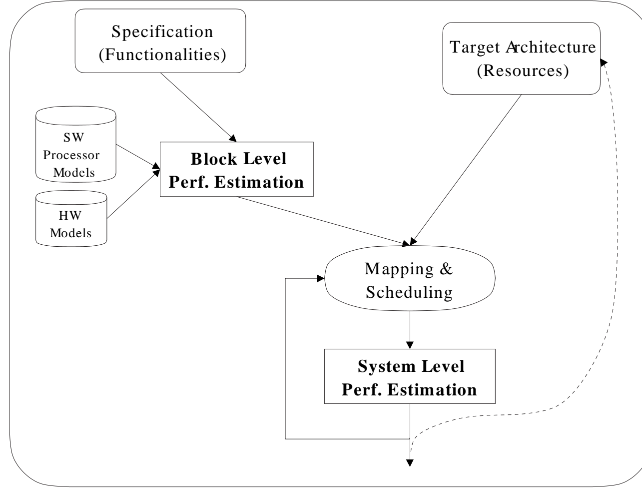

Figure 3.2: Design Space Exploration

algorithms for generating them:

$$S(n,k) = kS(n-1,k) + S(n-1,k-1) \tag{3.2}$$

Since a set of n elements can only be partitioned into 1 or n subsets in a single way,

$$S(n,1) = S(n,n) = 1 \tag{3.3}$$

A subset of these numbers can be seen in following Table 3.1 ([97]).

Moreover, if p different kinds of technology exist to implement each module (for example, each module may be implemented as specific hardware or targeted as a software executed on a specific processor), the number of possible architecture–partition alternatives (NA) increases, as shown by Formula 3.4.

$$NA(n,p) = \sum_{k=1}^{n} p^k S(n,k) \tag{3.4}$$

Observing the previous numbers and Formulas it is readily apparent that as the complexity of the system grows, the number of architecture–partition alternatives increases exponentially. For example, considering a system composed of 6 functions and where

Table 3.1: Stirling Numbers of the Second Kind

	k							
	1	2	3	4	5	6	7	8
n								
1	1							
2	1	1						
3	1	3	1					
4	1	7	6	1				
5	1	15	25	10	1			
6	1	31	90	65	15	1		
7	1	63	301	350	140	21	1	
8	1	127	966	1701	1050	266	28	1

2 different kinds of technology are used for each processing unit, the design space will contain

$$NA(6,2) = \sum_{k=1}^{6} 2^k S(6,k) = 2430 \quad (3.5)$$

possible architecture–partition solutions.

For each alternative, the construction of the model and the synthesis and low level co-simulation may take days. Thus, the synthesis and the simulation at the cycle level of every single architecture to measure its performance cannot be afforded. These facts constitute the basis of the motivation for the proposed methodology. This is the need for a system level performance estimation methodology together with an exploration strategy in order to narrow down the design space. The performance estimation method must support an easy (in terms of modelling effort), fast (in terms of simulation time) and accurate architecture exploration.

3.3 Performance Estimation

The estimation of performance plays an important role in design space exploration. It guides the selection of the target architecture as well as the partitioning and mapping of the functionalities onto the previously selected architecture. These decisions are made on the basis of various metrics which vary with design objectives and constraints.

Performance estimation involves two steps ([91]):

- Modelling of the system specification and the architecture: In this stage, the functionalities described in the system specification and the components which make up the architecture are represented in an abstract form, i.e. a formal representation or a sequence of instructions in any programming language;
- Evaluation of the performance: In this second stage, the models previously defined are processed and the related performance metrics extracted. The evaluation can be done in two different ways, i.e., analytical evaluation or evaluation by simulation. The analytical evaluation performs a sequence of transformations after which a set of equations are obtained. When using simulation, the models are executed with a time scale.

3.3.1 Performance Estimation Requirements

Performance estimation involves certain requirements concerning accuracy, modelling effort and evaluation effort. These issues depend on the level of abstraction. Within the context of design space exploration, additional issues have to be considered, as for example adaptability and generality of modelling and evaluation ([78]). A definition of each requirement and how they vary with the level of abstraction is provided below.

- Accuracy: Closeness of performance metrics evaluated by the estimation to the exact values of the real implementation. The accuracy increases with the detail grade of the modelling;
- Modelling effort: Cost and time to model the architecture and application. It decreases with increase in level of abstraction due to the fact that at higher levels of abstractions, fewer features of the components have to be modelled;
- Evaluation effort: Speed of evaluation. The evaluation effort increases with the detail grade of the modelling. Analytical evaluation is the fastest. On the other hand, if the evaluation is simulation-based, increasing the detail of the model means that the simulation kernel has to handle a larger number of events. This results in larger simulation times;
- Adaptability: Ability to change the level of detail of different models while keeping the framework fixed. It allows the performance of the evaluation at various levels of abstraction. A high adaptability implies less overall modelling effort for design space exploration;

- Generality of modelling and evaluation: Retargetability of modelling and evaluation. It can be said that a methodology is general, if during design space exploration, less effort is needed to change the system components or their internal architecture.

A further important requirement for a performance estimation method that is to be integrated in a design space exploration schema is the re-building effort, i.e., the effort necessary before a new alternative can be tested. The proposed methodology addresses this requirement by applying a graph whose structure does not have to be re-constructed each time a new alternative is explored.

Since communication is the main bottleneck in modern application domains ([92]), the performance of interconnections are crucial to the performance of the complete system. Therefore, the effect of performance degradation due to conflicts on interconnection should be accounted for in the system performance estimation. The proposed methodology also addresses this issue by taking into account the internal system communication.

3.3.2 Categorisation of Performance Estimation Approaches

In principle, the performance evaluation approaches can be classified based on the level of abstraction of the models and based on their evaluation methodology.
From the first point of view, at higher level of abstraction the models capture only timing details, whereas the functionality of the application and the architecture are either completely ignored or captured in less detail. When the level of abstraction is decreased, timing as well as functionality and architecture are captured in greater detail.
Considering the evaluation methodology, the performance evaluation approaches from analytical models (models at high levels of abstraction) can be further divided into two levels: evaluation by analytical methods and evaluation by means of simulation. And for the evaluation of models at lower levels of abstraction, different architecture simulators are utilised, such as simulators for instruction level models, cycle-accurate models and RTL and logic level models.

Figure 3.3 depicts where the different groups of performance estimation approaches mentioned above are introduced in the pyramid of models shown in Figure 3.1. Moreover, Figure 3.3 illustrates how the requirements concerning accuracy, modelling cost and execution time change with the different abstraction levels. Subsequently, these performance estimation methods are explained in detail.

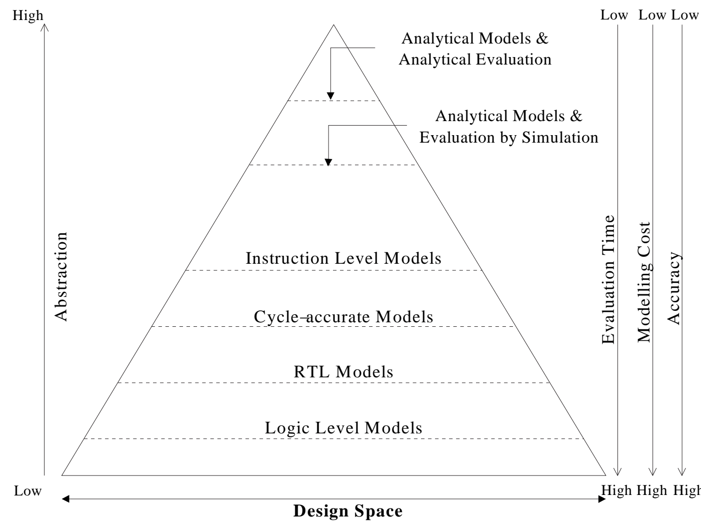

Figure 3.3: Performance Estimation Abstraction Levels

3.3.2.1 Analytical Modelling and Evaluation

At the highest level of abstraction, the models only describe functional behaviour and timing. A series of transformations is performed on such models in order to get a set of equations, which are solved to get the pursued performance numbers. These simplified models can be evaluated simply by solving mathematical equations and, therefore, the evaluation is very fast. Furthermore, the modelling effort is small and covers a wide range of design space. But, on the other hand, the accuracy of the extracted performance values is low because many details of the system are missing.

3.3.2.2 Analytical Modelling and Evaluation by Means of Simulation

The evaluation of analytical models of complex systems with greater detail is not possible with analytical techniques. Most of them suffer from state explosion problems or they do not account for resource contention. In these cases, the most frequent alternative to evaluate such analytical models is discrete event simulation. As simulation goes through a sequence of states one at a time, it can handle any lower level of abstracted models and deliver more accurate results. Nevertheless, most of the time simulation techniques are slow and speed depends on the abstraction level of the input models.

3.3.2.3 Instruction Level Models

At the instruction level, the detail of modelling the architecture allows the verification of the functional specification. The application code is translated into machine code instructions for the target processor. The executable instruction level model is simulated, carrying out the machine code instructions. The performance metrics are determined using fixed delay models for instruction execution and memory access. Therefore, these models provide more accurate performance results for single processor architectures but at the cost of simulation time. However, these models cannot describe the cycle boundary behaviour and its difficulties in modelling pipelining and instruction level parallelism. Moreover, the interaction and communication between the processor and other components within the architecture cannot be handled. Hence, instruction level models cannot be used for the performance evaluation of multi-processor systems.

3.3.2.4 Cycle-Accurate Models

Cycle-accurate models include micro-architecture level details. During simulation, machine-code instructions are executed one by one and events in each cycle can be traced. These models describe the correct functional behaviour and timing of the architecture at each cycle boundary, avoiding the constraints of the previous models concerning modelling of pipelining, parallelism and inter-component communication.

3.3.2.5 RTL and Logic Level Models

Register transfer level (RTL) and logic level models describe both functional and timing details and the logic level models can be realised in silicon. At this level, much more extensive information than in the previous models is found. For the performance evaluation of systems it is not necessary to reach the logic level because the RTL model suffices to have clock cycle-accurate detail of events.

In [78], the author shows a comparison of different architecture simulators working at instruction, cycle-accurate and RT level. SPIM is an instruction level simulator for MIPS 3000, tmsim is a cycle-accurate simulator for TriMedia architecture and DLX is an RTL model in VHDL. In order to compare the simulation speeds, it is assumed that a simple video algorithm takes 300 RISC-like instructions per pixel for one video frame of 720x576 pixels. The last column of Table 3.2 shows the simulation time spent by each of the three above-mentioned architecture simulators for processing one such frame.

3.4 Comparison of Performance Estimation Approaches

Every group of performance estimation approaches presented in the previous Section aim at determining how many times the system passes through the different states. What differentiates one approach from the other is the concept of state, which depends on the level of abstraction.

Next, the existing performance estimation techniques presented in Chapter 2.3 are classified according to the groups defined above and their advantages and disadvantages are discussed. Later, the hardware–software co-design procedures introduced in Chapter 2.2 are analysed in terms of inter-component communication support and implementation procedure. These two characteristics determine their accuracy and their potential for automation.

3.4.1 Advantages and Disadvantages

The analytical approaches model a simplified system and can be evaluated analytically or by means of simulation.
The analytical evaluation of analytical models aims at minimising modelling time and solution time, while providing sufficient prediction accuracy to reduce the design space. They are suitable for the initial design stages where prediction cost is of a higher priority than optimum prediction accuracy, given the huge design space involved. The static performance estimation methods presented in Section 2.3.2 belong to this group. In the case that the system is too complex or there is a need for detailed modelling, the simulation is one of the most suitable alternatives to evaluate such systems. The analytical performance estimation techniques introduced in Section 2.3.2 fall into this category. For instance, the performance simulation language PAMELA ([77]) contains constructs for capturing dependencies, resource contention, execution delay and con-

Table 3.2: Comparison of Architecture Simulators

Simulator	Architecture	Accuracy	Sim. Speed	Performance
SPIM	MIPS 3000	instruction level	200000 ins./sec.	10 min.
tmsim	TriMedia	cycle accurate level	40000 ins./sec.	54 min.
DLX	DLX	RTL level	500 ins./sec.	1.2 days

ditional control flow. If the model under study has just the three first features, then it can be evaluated analytically, otherwise simulation is required. PAMELA has been used in [76] for the implementation of an analytic performance modelling approach, while in [78] PAMELA has been applied in design space exploration for stream-based data flow architectures following a simulation-based approach.

The simulation-based approaches presented in Section 2.3.2 use either simulators for instruction level, cycle-accurate or RTL models. The simulators for instruction level models are not able to handle multi-processor systems due to the fact that timing details are only available at instruction boundaries. In contrast, the simulators for logic level models make it a very time consuming task to run software on it. Therefore, the most commonly used approach is the cycle-accurate simulation, where many logic level details are omitted to speed-up the simulation. Nevertheless, the cycle-accurate simulation is also a time consuming approach.

Lastly, the trace-based techniques introduced in Section 2.3.2 apply both analytical and simulation-based methods. They extract from the initial specification the information about the computations and communications of the system components. With this information and the parameters of the system resources, such techniques deliver an estimation of the final performance. Trace-based techniques are suitable for cases where the initial co-simulation is not easily reproducible. The extraction of traces from the initial co-simulation has direct influence on the inspection of diverse architecture combinations. Nevertheless, the traces to be extracted are fixed in advance as well as their sequence. A change during the evaluation phase is not possible.

3.4.2 Support of Co-Design Procedures

Analysing the co-design procedures presented in Chapter 2.2 in terms of inter-component communication support and implementation process, and considering the comparative Table 2.1, it can be concluded that the communication aspect of the estimation of the performance is only covered by two of the approaches, Chinook ([93]) and SpecC ([19]). In both cases the partitioning process is done manually. Only by Cosyma ([94]) has the partitioning process been automated. However, the communication is only taken into account locally, without considering the influence of other units, a factor which could drastically modify the final performance results.

After the refinement of the architectural model, the SpecC methodology provides accurate estimations for the whole range of software and hardware components. But an estimation of the time a selected bus is used by a given communication channel can only be extracted after communication refinement. The partitioning of the functions has already to be determined in the previous stage, without considering the communication

effects.

In Chinook, the design space exploration is enabled by applying single system specifications that capture the reactive real time behaviour of the system. At the higher level, Chinook facilitates easy migration of functionality among processing elements and manages the communication requests between the processors. This enables designers to rapidly evaluate different architecture partitioning. However, these decisions require user interaction.

Hardware and software performance estimations, but no communication estimations, are extracted following the hardware and software synthesis respectively and are used by Polis ([67]) during the partitioning step. It makes this procedure unsuitable for systems where access conflicts to shared elements occur often. Nevertheless, recently, related works have been presented, such as [95], which studies the effects of shared memory buses during system level performance analysis in the Polis co-design environment.

Finally, concerning the existing system level development environments also presented in previous Chapter, it can be concluded that they are prepared for testing different hardware–software architecture–partition alternatives. Nevertheless, this exploration becomes a time consuming task due to the extensive customisation required and the necessary rebuilding of the structure in case new building blocks have to be added. Furthermore, often they require a high modelling effort (for example, in VCC four models are necessary for one design) and own a tool-specific description language.

3.4.3 Need for Improvements

After presenting the disadvantages and deficiencies encountered in the existing approaches for performance estimation and in the co-design procedures, it can be concluded that there is a need for a fast performance estimation methodology that captures architecture (including the communication aspect) as well as application behaviour in order to speed-up the design space exploration. For achieving this purpose, the methodology has to support a fast (in terms of simulation time and re-building effort) exploration of various design choices as well as variations in the target architecture. It also has to support the modelling of inter-component communication and simultaneous access conflicts which arise in multi-processor architectures. Moreover, these goals have to be achieved without losing much accuracy in the extracted performance values. Finally, an automation of the whole estimation procedure would allow an easy and faster design space exploration.

The method proposed in the present thesis covers the requirements mentioned above.

The adopted approach is based on a graph model which is evaluated by means of simulation. The new methodology consists of a two-step procedure. First, the system is modelled in terms of a graph, where the relevant information concerning the target architecture is annotated and a mechanism to solve resource contentions is added. This functional modelling accelerates the exploration of further alternatives without requiring a re-building of the structure if the hardware structure is changed. Second, the evaluation is performed by means of simulation at the system level. It allows a more detailed analysis than analytical evaluation. And, at the same time, the simulation runs faster than a simulation at a lower level of abstraction.

3.5 Integration in the Design Flow

In Figure 3.4, the positioning of the proposed performance estimation methodology within the system level design flow is depicted. It follows the framework presented in Figure 3.2 for design space exploration at system level.

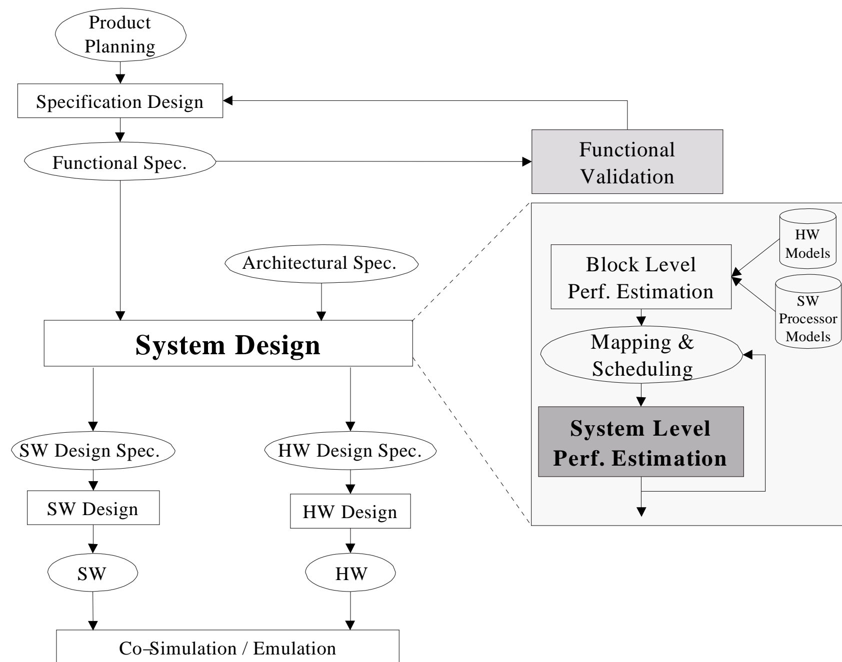

Figure 3.4: Integration Performance Estimation Process in the Design Flow

The first step in the design flow is the description of the system functionality in terms of an executable functional specification. This description is then functionally

validated and, when no changes in the specification are needed, the architecture exploration phase begins. This stage comprises the selection of the target architecture and the partitioning of the functionalities. Furthermore, the partitioning includes the following steps: mapping; hardware sharing; scheduling; interfacing; and pipelining, which cannot be considered independently but are related to each other. A decision concerning these parameters is taken and, along with the system description and the selected target architecture, they constitute the inputs for the performance estimation procedure. In case the performance results for the tested architecture–partition alternative do not meet the predefined performance constraints, a new partition and/or target architecture has to be tried. The process flow remains in this loop as long as the constraints are not satisfy. Once the architecture–partition alternative that fulfills the requirements is found, the further steps in the design flow are performed. Then, the hardware, software and interface parts are designed, followed by the co-synthesis phase and the back-end.

The framework for the estimation of the performance achieved by a certain alternative is divided into two stages: Block Level performance estimation and System Level performance estimation. In the first stage, the performance values for each subfunctionality mapped to each processing unit are calculated. This task is performed with the help of hardware and software processor models. These values are then stored in an internal library which is accessed by the System Level performance estimation. In this second stage, the processing time of each sub-functionality is annotated, depending on the processing unit to which it is mapped to. Moreover, the conflicts when accessing the shared elements are solved, the transfer time through the communication medium is annotated and the related queues are simulated.

The methodology proposed in this thesis covers the second stage, the System Level performance estimation. Nevertheless, the results of the Block Level performance estimation are required as inputs before performing the latter stage. The next two Chapters 4 and 5 explain the procedure and implementation of the proposed methodology in detail.

3.5.1 Inputs Required by the Performance Estimation

As depicted in Figure 3.4, the proposed performance estimation methodology requires as input the information concerning the system specification, the allocated target architecture, the results of the partitioning phase and the Block Level performance estimation values.

The system specification is first of all divided into a set of smaller pieces, so-called granules. In the proposed methodology, a decision concerning the granularity of the

functions has to be predefined. A compromise between fine coarse granularity (function level, command level or instruction level) has to be reached and used further on in the definition and description of the single functionalities. When choosing the granules, the memory organisation has to be taken into account due to the fact that internal transfers are only possible when a function has finished execution.

On the other side, a target architecture for the implementation is selected. In the hardware–software partitioning phase, a mapping of the previously defined granules to hardware or software is performed. The aspects to be considered when dealing with the hardware–software partitioning problem are as follows:

- Hardware–software mapping: Decision regarding the processing unit in which a function is executed. The key aspect of this decision is that a processor can only execute a single function at a time, while a hardware unit can be synthesised to execute multiple functions concurrently;
- Hardware sharing: Minimisation of the hardware area by sharing hardware resources for different units if possible;
- Interfacing: Insertion of the required communication between two consecutive functions mapped to two different processing units;
- Scheduling: Determination of the order in which the functions are fulfilled. The scheduling has, for instance, to guarantee that a processor is only occupied by one function at a time. The execution of functions sharing the same hardware resources has to be scheduled as well. Finally, all transfers on each communication channel have to be sequentialised;
- Functional pipelining: Restart of the system for new incoming data although the old inputs have not yet finished execution. Using functional pipelining, the overall system execution time of the schedule can be optimised.

Most of these aspects are interacting (e.g. hardware sharing and scheduling; two functions sharing the same hardware resource have to be sequentialised) and, therefore, they have to be considered together. A decision concerning these parameters is taken and used further on.

3.5.2 Boundary Conditions

Considering the inputs of the methodology presented in the previous paragraph, some boundary conditions have to be taken into account when developing the new method-

ology. Such boundary conditions are introduced and explained below.

The proposed methodology is oriented to support the design of multi-processing, multi-threading SoC networking architectures. Architectures such as these, for instance the packet processors, are intended to deal with the user plane functions (no control/management plane functions) at the edge of the network, where high speed is required.

These architectures contain, mainly, multiple RISC embedded processors, powerful co-processors to accelerate some costly functions and memory and interface blocks. Different communication resources connect all building blocks with each other. A bus-based communication is very often found. An important characteristic of the RISC embedded processors is its multi-threading hardware support to avoid idle times when waiting on results from other resources. The modelling and evaluation support of this multi-threading characteristic also constitutes a novelty aspect of the proposed method, which has not been previously covered by other performance estimation approaches.

Furthermore, an abstract bus-based System-on-Chip architecture is taken as the basis for the development of the methodology. Communication among units takes place through one or several shared buses and point-to-point connections are supported as well. For the applicability of the methodology to other kinds of communication architectures, as for instance a ring, it would be necessary to adapt the arbitration mechanism of the buses.

Figure 3.5 shows the abstract target architecture taken as a basis for the development of the proposed methodology. It contains several processing units (PU_k), memory units (M_p) and the required interface units to the peripheries. The processing units can be either low level RISC (Reduced Instruction Set Computer) processors (PE_n) with multi-threading capability and no RTOS (Real Time Operating System) running on them, or hardware blocks (AC_m) accelerating certain functions. The memory units can be either global or local to specific blocks. The data communication among the different units takes place through a common bus (one or several dedicated units), except for the access to the local memories. When more than one master attempts to initiate an access to the common bus, an arbiter is necessary to manage the requests.

The methodology contains constructs for capturing dependencies, resource contention, execution delay and conditional control flow. Therefore, it is applicable to systems having multiple processors as well as application-specific hardware units. Moreover, hardware multi-threading embedded processors, which are often found in networking packet processors, are supported as well.

For the parameters related to the partitioning phase, which are required as inputs for the performance estimation method, the following assumptions are made. First of

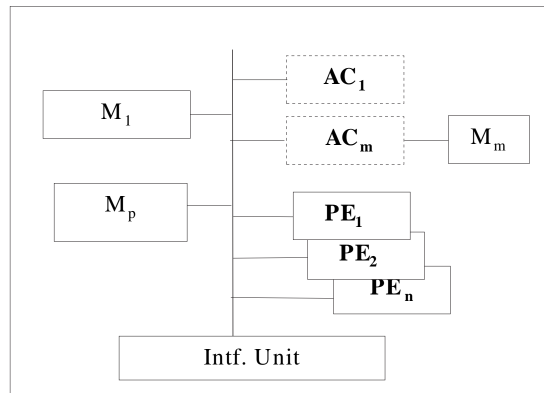

Figure 3.5: Target Architecture

all, the mapping of the functionalities onto the selected target architecture is taken as input. Second, because of the predefined bus-based communication architecture, every slave processing unit attached to a certain bus is shared by the master processing units linked to the same bus. Third, the borders between processing units are detected and, between the functions concerned, a mechanism to access the shared communication medium binding both processing units is introduced. Fourth, an ASAP (As Soon As Possible) scheduling algorithm is assumed, i.e., the functions executed as soon as practicable while maintaining the constraints imposed by the other aspects related to the partitioning phase. Fifth and last, it is assumed that functional pipelining is not feasible and, moreover, the hardware units do not support pipeline execution.

Chapter 4

System Level Performance Estimation for Multi-Processing, Multi-Threading SoC Architectures

4.1 Chapter Introduction

The present work introduces a novel system level performance estimation methodology which supports a fast design space exploration based on a functional graph. The rebuilding effort is considerably lower when applying the proposed methodology rather than building up a structural model of the target architecture at a lower level of abstraction. For the modelling of such a graph, a novel usage of the system level language SystemC has been applied.

This Chapter is organised as follows: First, the fundamentals of the methodology are introduced and the selected abstract target architecture is depicted. The modelling and evaluation approach applied for the development of the proposed methodology is then presented. Last, the system performance estimation scheme is depicted and the related issues are explained in detail.

4.2 Fundamentals of the Methodology

Figure 4.1 illustrates two different approaches that can be followed when pursuing a first estimation of the performances achieved by a concrete architecture–partition alternative. On the one hand, a structural model of the target architecture is built up

according to the selected partition of the functionalities. It delivers a precise estimation, but most of the time the effort and time it takes to try a new alternative is considerable when a change of the hardware architecture is required. On the other hand, the functionalities are described in terms of a formal representation, such as a deterministic graph, and the relevant information of the target architecture is added to the graph. This information comprises, first, the performance of each function mapped to the selected processing unit and, second, the resolution of resource conflicts. In this case, the simulation time and modelling effort towards a re-partitioning is less costly because the structure of the graph does not have to be rebuilt each time a new architecture–partition alternative is to be tested. Consequently more partition alternatives can be simulated and evaluated. After choosing a partitioning that meets the performance constraints, the path towards synthesis (back-end) goes through a structural model of the target architecture. This step is costly in terms of design, but fortunately it has to be done only once after selecting the best partition alternative that meets the constraints through the loop described above. The proposed methodology introduces the second approach. A functional model based on a graph representation endorsed with a mechanism to solve resource contentions is used for the estimation of the performance instead of a structural model.

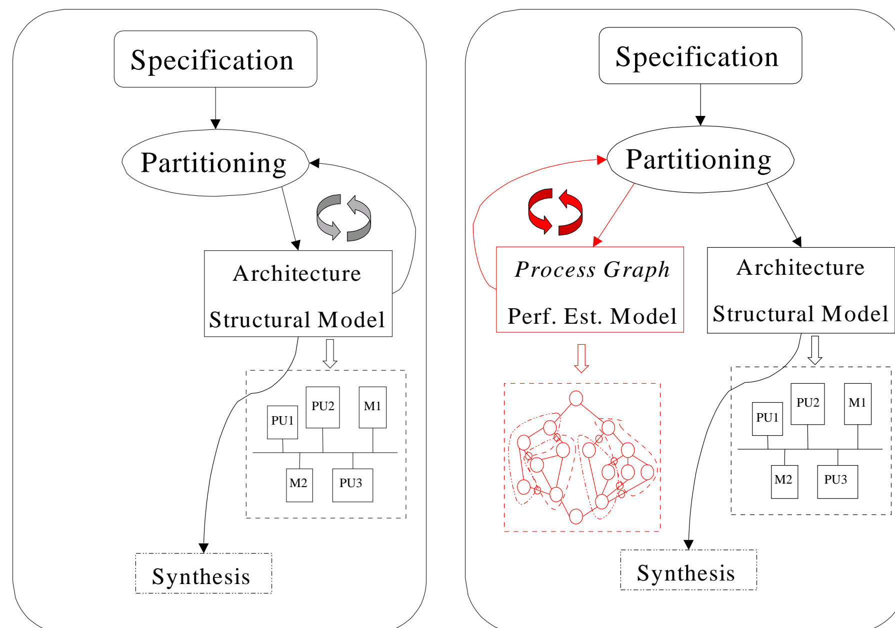

Figure 4.1: Estimation of Performance

4.3 Modelling Approach

The main application scenario of the present thesis, i.e., networking architectures, is control-dominated. The further processing of the packets crossing in a networking equipment depends on the values of certain fields and on external configuration information. This kind of system is suitable to be modelled using a Conditional Process Graph (CPG), which is able to capture both data and control dependencies. That is the main reason for choosing the CPG proposed by Petru Eles ([98]) as the basis for developing the proposed methodology. Further graph-based representations that also capture data and control dependencies, like the Control Data Flow Graph (CDFG) presented in [56], are mainly intended for application scenarios built up around infinite loops, which is not the case of the networking data flow.

Petru Eles uses an abstract model for system representation based on a directed, acyclic polar graph. Each node of the graph represents a process. Such a process can either be a *computation process* specified by the designer or a so-called *communication process* which captures the message-passing activity. The edges of the graph are either simple or conditional edges. A conditional edge has one associated condition value. Transmission on such an edge takes place only if the associated condition value is true and not, like on simple edges, for each activation of the input process. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and last process. Now, considering an architecture consisting of several processing units connected through buses, each *computation process* of the graph is assigned to a processing unit and each *communication process*, which connects *computation processes* assigned to different resources, is assigned to a communication channel. A mapped process graph is thus built up.

For the modelling of the graph, the industry standard system level language SystemC has been chosen. It provides flexibility of modelling at various levels of abstraction and performs faster simulation. SystemC also helps in modelling timing, reactivity and concurrency and in evaluating resource contentions. Such characteristics make this language especially suitable for describing the initial Conditional Process Graph and, later on, for annotating the timing information and inserting additional nodes which model the internal system communication and the required arbitration and synchronisation mechanisms.
In the proposed methodology based on the CPG and employing the support of SystemC, an Annotated SystemC Conditional Synchronisation Graph (ASCSG) ([99]) is built. It comprises three phases, the Functional, Architectural and Communication phase, which are built one on top of the previous one.

The definition of the three phases which build up the ASCSG and their main characteristics are presented next ([100]).

4.3.1 Functional Annotated SystemC Conditional Synchronisation Graph (Functional ASCSG)

The Functional ASCSG contains the description of the system functionalities in form of a Conditional Process Graph (CPG). The initial implementation of the graph, shown in Figure 4.2, includes *computation nodes* (Cp_i). They represent the single processes into which the system functionality has been divided. The edges of the graph are either simple or conditional edges. A conditional edge has one associated condition value (K_j). Transmission on such a node takes place only if the associated condition value is true and not, as on simple edges, for each activation of the input process.

Before starting the description of the Functional model, a decision concerning the granularity of the *computation nodes* has to be taken. A compromise between fine coarse granularity (function level, command level or instruction level) has to be reached and the selected level has to be used further on in the definition and description of the *computation nodes*.

Each *computation node* is modelled as a SystemC module. Each module comprises the node functionality and the required input and output ports. The conjunction of such modules in the predefined sequence describes the complete system functionality.

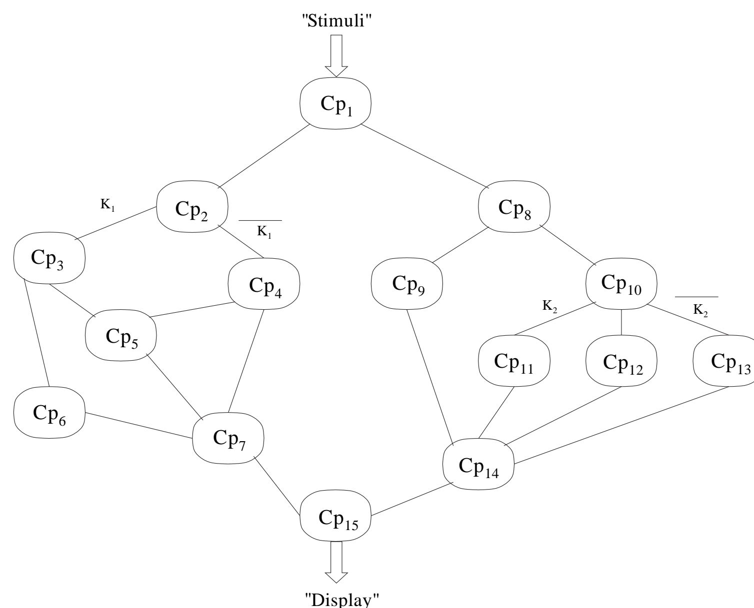

Figure 4.2: Functional ASCSG

4.3.2 Architectural Annotated SystemC Conditional Synchronisation Graph (Architectural ASCSG)

The Architectural phase models the mapping of each *computation node* (Cp_i) onto a processing unit (PU_k) which belongs to the selected target architecture presented in Figure 3.5. Furthermore, in the case of a multi-threading software implementation, the *computation nodes* are mapped to the selected threads ($\mu thr_{t\text{-}n}$) inside a multi-threading embedded processor.

The result of this mapping is the annotation of each *computation node*, CP_i, with its execution time, ($t[CP_i/PU_k]$), depending on the processing unit, PU_k, on which it is mapped. The internal library, where the results of the Block Level performance estimation are stored, provides the performance data of each block mapped to the selected processing unit.

In Figure 4.3, a mapping alternative for the previous Functional ASCSG (Figure 4.2) is depicted. For instance, the *computation nodes* CP_2, CP_3 and CP_6 are mapped to the processing unit PU_1. PU_1 is a multi-threading embedded processor and therefore the *computation nodes* mapped to this processor are further assigned to the corresponding threads. In this case, the CP_2 and CP_3 are mapped to one thread and the CP_6 to a second thread.

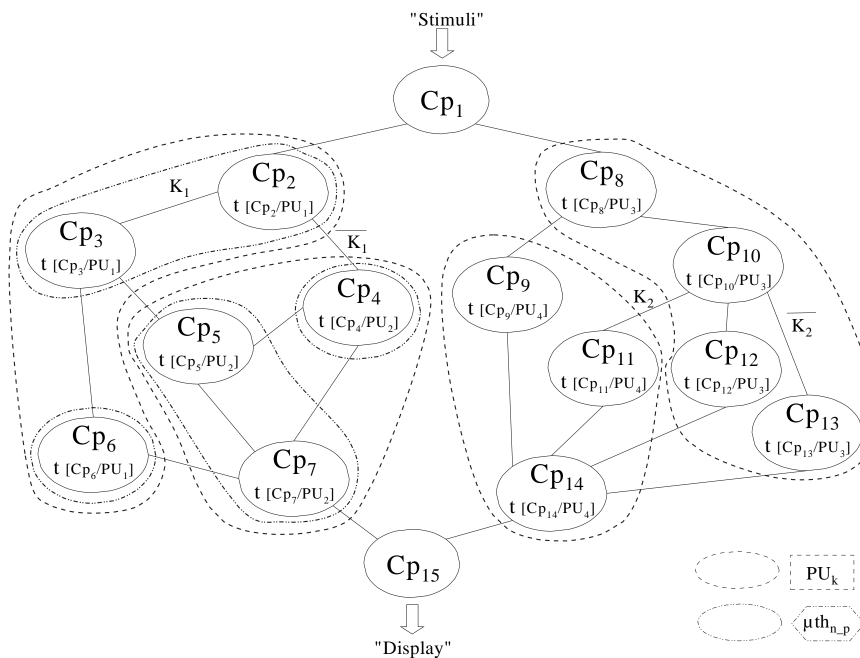

Figure 4.3: Architectural ASCSG

After defining the mapping of the *computation nodes* onto the processing units, the *computation nodes* which are mapped to the same processing unit and are triggered by diverse $\mathrm{Cp_i}$ running in parallel have to be detected. For accessing such nodes, an arbitration mechanism to store the requests and process them in a predefined order has to be provided. Such a mechanism is referred to as a *shared element* mechanism ($\mathrm{Shel_j}$). In order to send the access-requests to the $\mathrm{Shel_j}$ an *access node* ($\mathrm{Acc_{i\text{-}j}}$) has to be interleaved between the shared *computation node* and the node which attempts to access it for writing or reading. Such $\mathrm{Acc_{i\text{-}j}}$ is attached to the corresponding $\mathrm{Shel_j}$ which keeps the requests until the *shared element* mechanism grants access to the shared *computation node*, as shown in Figure 4.4.

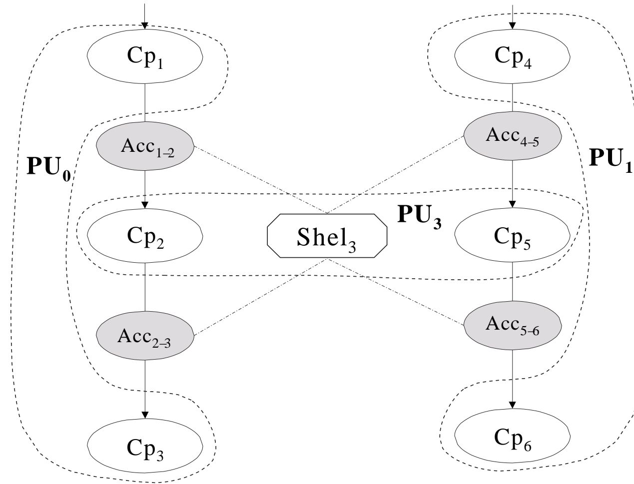

Figure 4.4: Shared *Computation Node*

4.3.3 Communication Annotated SystemC Conditional Synchronisation Graph (Communication ASCSG)

In order to cover dynamic effects of the system internal communication, additional nodes, the so-called *communication nodes*, are introduced, which capture the message passing activity. They can either be *external communication nodes*, i.e., *communication nodes* inserted between *computation nodes* mapped to different processing units (e.g., $\mathrm{Cme_{i\text{-}j}_m_y}$ is inserted between $\mathrm{CP_i}$ and $\mathrm{CP_j}$ which are mapped to different $\mathrm{PU_k}$s); or *internal communication nodes*, i.e., *communication nodes* inserted between *computation nodes* mapped to different threads within a multi-threading embedded processor (e.g., $\mathrm{Cmi_{i\text{-}j}_n_t}$ is inserted between $\mathrm{CP_i}$ and $\mathrm{CP_j}$ which are mapped to different $\mu\mathrm{thr_{t\text{-}n}}$s). Each $\mathrm{Cme_{i\text{-}j}_m_y}$ is assigned to a Communication Medium, $\mathrm{m_y}$, whereas each $\mathrm{Cmi_{i\text{-}j}_n_t}$

is assigned to a multi-threading embedded processor, PE_n.

For the modelling of the communication between processing units and the synchronisation between threads, two arbitration mechanisms are required. The Command Bus Arbiter (CBA_y) arbitrates the access to the shared communication media whereas the Event Arbiter (CEA_n) synchronises the threads within a multi-threading embedded processor. A CBA_y is added for each communication medium, m_y, and the corresponding *external communication nodes* ($Cme_{i\text{-}j}_m_y$) are attached to it. Furthermore, a CEA_n is added for each multi-threading embedded processor, p_n, and the corresponding *internal communication nodes* are attached to it ($Cmi_{i\text{-}j}_n_t$). Both arbitration mechanisms implement a management policy which has to be provided. By default, a priority-based policy for the CBA and a round-robin arbitration scheme for the CEA are supported. These elements can be seen in Figure 4.5 where a section of the Communication ASCSG which enhances the previous Architectural ASCSG (Figure 4.3) is depicted.

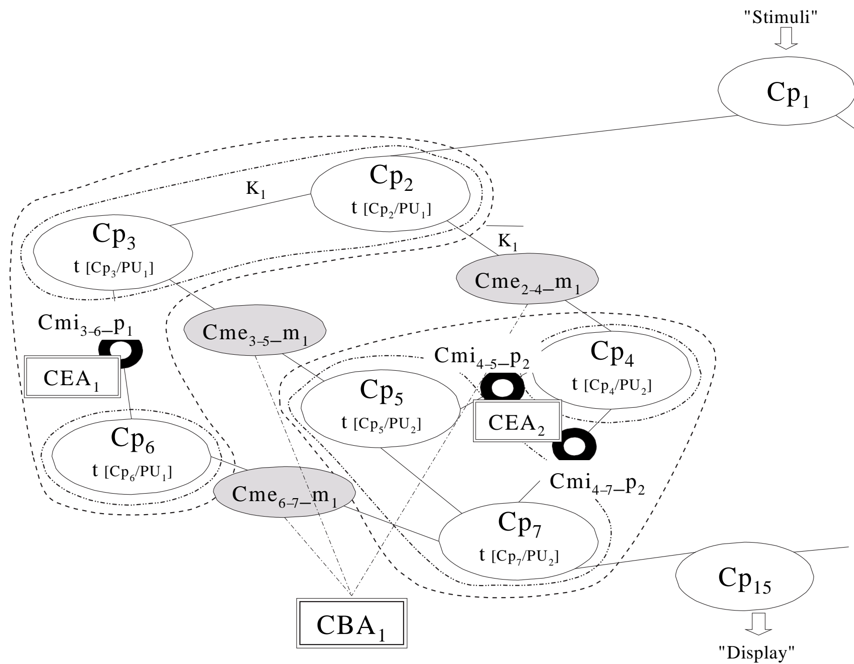

Figure 4.5: Communication ASCSG

4.4 Evaluation Approach

The previously modelled Annotated SystemC Conditional Synchronisation Graph - (ASCSG) is evaluated by means of simulation. It allows a more detailed analysis than analytical evaluation and, at the same time, the simulation runs faster than a simulation at a lower level of abstraction.

The initial model of the graph, the Functional ASCSG, which only contains the functional description of the system specification, is simulated in order to provide the correctness of the input specification.

The simulation of the Architectural ASCSG determines the parallel/sequential processing of nodes, taking into account the restriction imposed by the mapping and the scheduling information, i.e., the determination of the sequence in which the operations are fulfilled. By default, an ASAP scheduling procedure is supported, in which each operation is performed as soon as the previous one has been completed. The only restrictions imposed are the ones concerning the access to shared resources. Each initiator owns a priority number which will grant it access on the basis determined by the management policy implemented by the corresponding shared resource.

Finally, the evaluation of the Communication ASCSG cover the system internal communication, i.e., the communication between processing units and the synchronisation between threads within a multi-threading embedded processor. The activities that are considered in both cases are explained below.

For the case of a bus-based communication architecture, the following aspects are considered during a data transmission:

- The initiator processing unit first signals its intention to the Command Bus Arbiter (CBA_y) through the corresponding *external communication node* ($Cme_{i\text{-}j}\text{-}m_y$). It will grant access to the target communication medium;
- A management policy for each CBA_y has to be selected. A default priority-based policy is implemented.

During an inter-thread communication the following is taken into account:

- The context/s that are ready to run signalise their status to the Context Event Arbiter (CEA_n) (waiting for a context switch event) through an *internal communication node* ($Cmi_{i\text{-}j}\text{-}n_t$);
- Once the active thread performs a context switch, this command is signalised to the CEA_n through the corresponding $Cmi_{i\text{-}j}\text{-}n_t$;

- A management policy for each CEA_n has to be selected in order to decide which is the next thread to run.

The simulation of the Communication ASCSG delivers the performance characteristics required for the evaluation of a concrete architecture–partition alternative. Some examples of the performance values that can be extracted from the simulation are the following: processing time; delay in accessing each shared communication medium; arbitration delay; delay in accessing each shared processing unit; load of each shared processing unit; delay accessing the process unit by a thread; and the context switch delay. For the extraction of further performance values, the methodology can be easily adapted.

For these evaluation purposes, a test-bench is built around the graph model. This test-bench feeds the graph with stimuli and displays the simulation results of the graph, as can be seen in Figures 4.2, 4.3 and 4.5.

4.5 System Performance Estimation Scheme

The estimation of the performance of a certain solution (i.e., a certain mapping–scheduling alternative of a system specification onto a selected target architecture) is carried out following the procedure depicted in Figure 4.6. As mentioned above, it comprises three phases, the Functional, Architectural and Communication phase, which are built one on top of the previous one. On the left side, the inputs of each phase are shown, i.e., the required information for the modelling of the graph. And on the right side, the outputs of each phase are depicted, i.e., the evaluation of the graph performed in each phase.

4.5.1 Inputs for the Modelling of the ASCSG

The single processes which make up the complete system functionality are selected from a library of functions. Such a library is created by the designer and contains the description of each process in form of a SystemC module. Such modules are then linked in the predefined sequence in order to build up the Functional ASCSG. The test-bench functions required for the further evaluation of the graph are additionally defined and bound to the graph for feeding the model with stimuli and displaying the results.

The inputs required by the Architectural ASCSG are the following. First of all, the resources (types and number of processing units, memories and interface units) of the target architecture have to be provided. Furthermore, in case of embedded processors,

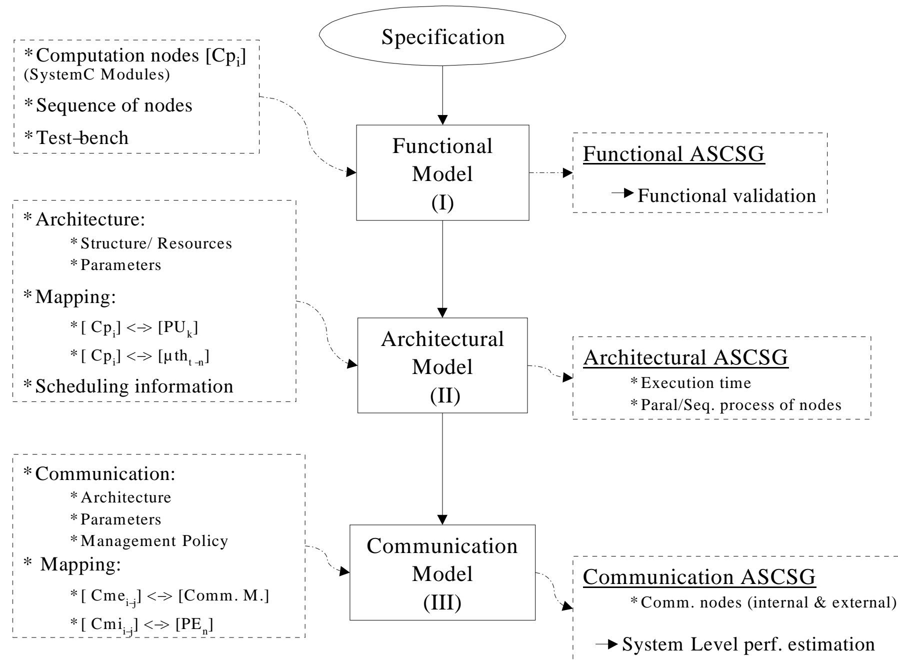

Figure 4.6: System Performance Estimation Scheme

it has to be defined whether they support multi-threading and, in the affirmative case, the number of threads supported. Secondly, the information concerning the mapping of *computation nodes* onto the resources and threads has to be defined. Lastly, the predefined scheduling of the *computation nodes* is also taken as input in order to define the overall flow of functions.

The last phase, the Communication ASCSG, requires as inputs the structure of the communication architecture, the related parameters for each communication medium (bandwidth and frequency) and the management policy implemented when accessing a shared communication medium and when sharing a processing unit by several threads. Furthermore, the mapping of the *communication nodes*, the external ones onto the communication media and the internal ones onto the multi-threading embedded processors, is necessary.

During the modelling of the ASCSG, different issues have to be considered. The definition of each issue and the proposed approaches to model them in SystemC are presented below.

4.5.1.1 Sequence of Nodes

The *computation nodes* in the Functional ASCSG are triggered by events, either input data changes or rising/falling edges in case of boolean inputs, as can be seen in Figure 4.7. The sensitivity list of each process inside each *computation node* defines the signals or ports which trigger the corresponding process. In this way, the process graph which makes up the system specification is built up.

4.5.1.2 Convergence of Paths

Within a process graph, it can occur that the outputs of two or more *computation nodes* sourcing from different branches which are not exclusive to each other meet together. At this point, it cannot be assured that the upper nodes deliver the outcome with the same latency. Therefore, the meeting node has to wait until all required inputs are available. In order to avoid the loss of intermediate results, a *fifo channel* (SystemC primitive channel `sc_fifo`), as depicted in Figure 4.8, has to bind each upper node with the meeting one. This case has to be taken into account when modelling the Functional ASCSG.

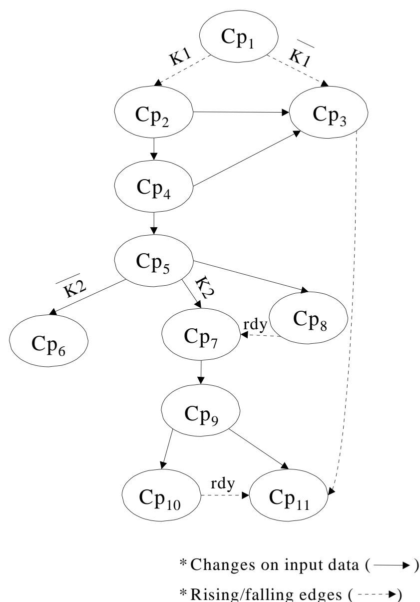

Figure 4.7: Sequence of Nodes

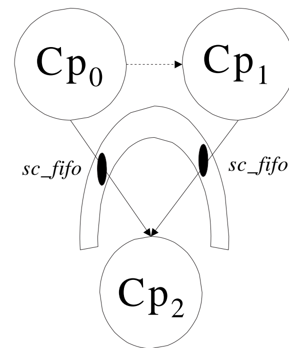

Figure 4.8: Convergence of Nodes

4.5.1.3 Flexible Mapping

The proposed technique allows the exploration of different mapping–scheduling alternatives without requiring a re-building of the graph structure. This is possible owing to a flexible mapping of the *computation nodes* onto the hardware units and onto processors and threads, Figure 4.9, which belong to the target architecture. The annotation of each *computation node* with the respective execution time (`x`), depending on the processing unit on which it is mapped, is carried out using the SystemC statement `wait(x,SC_NS)`. The usage of timing control statements is only possible in `SC_THREAD` or `SC_CTHREAD` processes. Hence, the processes involved have to be defined as `SC_CTHREAD` if they are sensitive to an edge of a clock or as `SC_THREAD` in the general case. At this stage, special care must be taken because multiple interacting synchronous processes must have at least one timing control statement in every path.

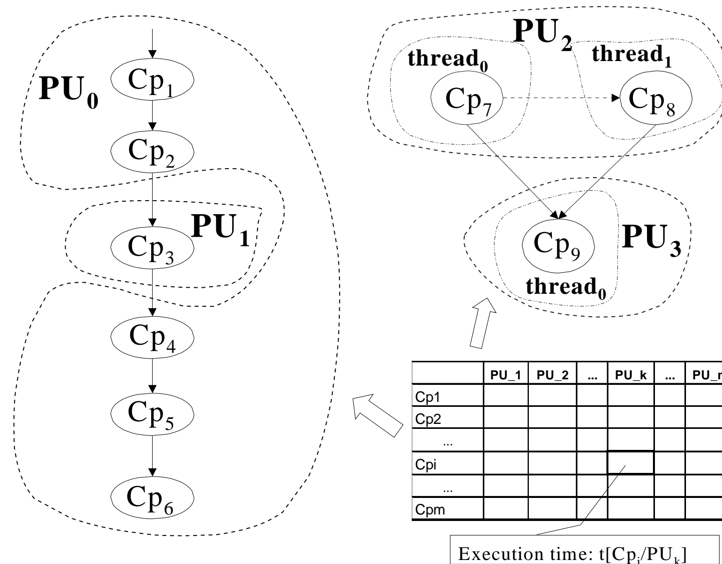

Figure 4.9: Mapping onto PE/Thread–AC

4.5.1.4 Multiple Instances of *Computation Nodes*

In the case of multiple processing units or threads performing the same functionality, as shown in Figure 4.10, the corresponding *computation nodes* are defined only once and instantiated several times. In SystemC, this can be performed by defining a pointer to the related *computation node* ($\texttt{Cp}_\texttt{i}$`*`) and making several instances of the node in the main file, which can be customised at elaboration time by passing constructor arguments.

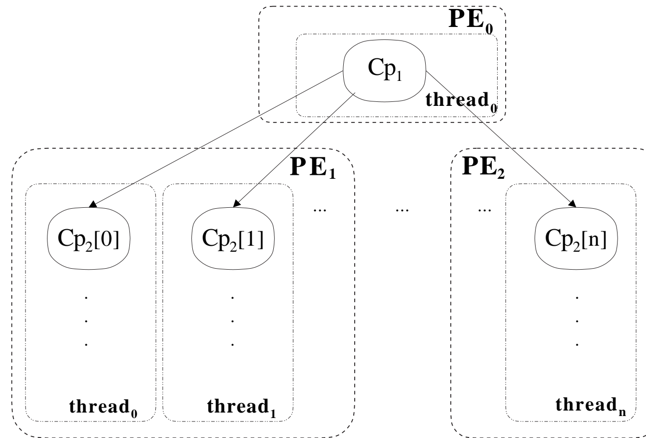

Figure 4.10: Multiple Instances of *Computation Nodes*

4.5.2 Outputs of the Evaluation of the ASCSG

The evaluation of each of the three models, the Functional, the Architectural and the Communication ASCSG, is performed by means of simulation. The outputs of these simulations are presented below.

4.5.2.1 Functional Validation

The evaluation of the Functional ASCSG results in its functional validation, which provides the correctness of the specification. For this purpose, a test-bench is built around the model, which can be reused in further models later on. The test-bench (Figure 4.11) comprises a module to generate the stimuli and a module to visualise the results.

4.5.2.2 Processing of Nodes

The evaluation of the Architectural ASCSG results in the annotation of each *computation node* with its execution time and in the determination of the parallel/sequential processing of nodes.

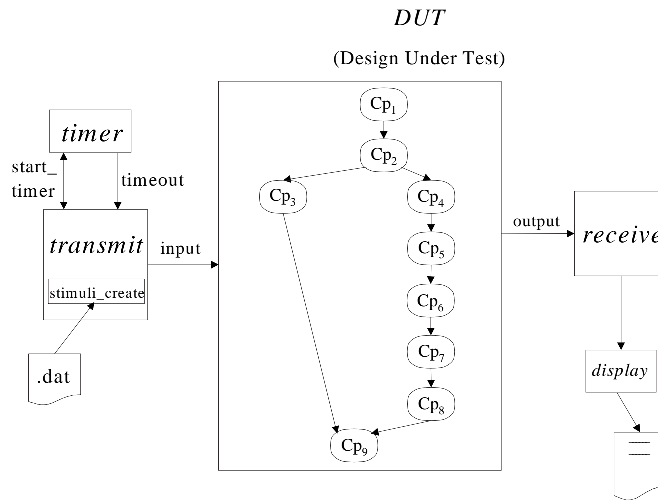

Figure 4.11: Functional Validation

4.5.2.3 Tracking and Monitoring

From the simulation of the Communication ASCSG, several performance values are extracted whose evaluation determines the performance estimation of the system. The proposed methodology is prepared to deliver such values, which can be very useful for the designer to analyse each alternative. The analysis of the results is then a task for the designer, who uses this methodology for exploring different architecture–partition alternatives, but not a task of the methodology itself.

Some examples of such values are: processing time of each processing unit; delay accessing each communication medium; arbitration delays; delay accessing each shared processing unit; load of each shared processing unit; delay accessing the process unit inside each embedded processor by each thread; and context-switch delays. In case the designer requires additional performance values for which the proposed methodology is not a priori configured, the designer can access the provided code for the arbitration mechanisms and call the *value_extraction()* function for the required parameters.

Chapter 5

Implementation

5.1 Chapter Introduction

The exploration of several architecture–partition alternatives and even the inspection of different target architectures is facilitated, provided that the user interaction can be automated. Using configuration files, the method is fed with the required input parameters. Then, a parser analyses the syntax of the corresponding configuration file and lastly a generator creates the related SystemC file. These files are further compiled and linked to the SystemC core and specific libraries in order to generate the pursued executable models, the Functional, Architectural and Communication model.

First, this Chapter shows the implementation procedure for the creation of the Functional, Architectural and Communication ASCSG. Subsequently, the data structures and configuration files used when pursuing the automation of the method procedure are presented.

5.2 Implementation of the ASCSG

The procedure towards the construction of the Annotated SystemC Conditional Synchronisation Graph is depicted in Figure 5.1. A three-step procedure has been developed for the modelling of the Functional, Architectural and Communication model. The first column shows the required input parameters, files and templates for each model. The supply of such parameters constitutes the first step of the procedure. A generator takes this information and creates the corresponding configuration files. Each of them takes into account the configuration file of the previous more abstract

graph. In the second stage, these configuration files are then parsed in order to generate the corresponding SystemC file. Finally, these SystemC files are compiled with a C/C++ compiler and linked to the SystemC core and specific libraries. In this way the executable files are generated whose simulation deliver the functional validation of the specification (simulation of the *main_functional.o*) and the extraction of the performance values for the selected alternative (simulation of the *main_communication.o*).

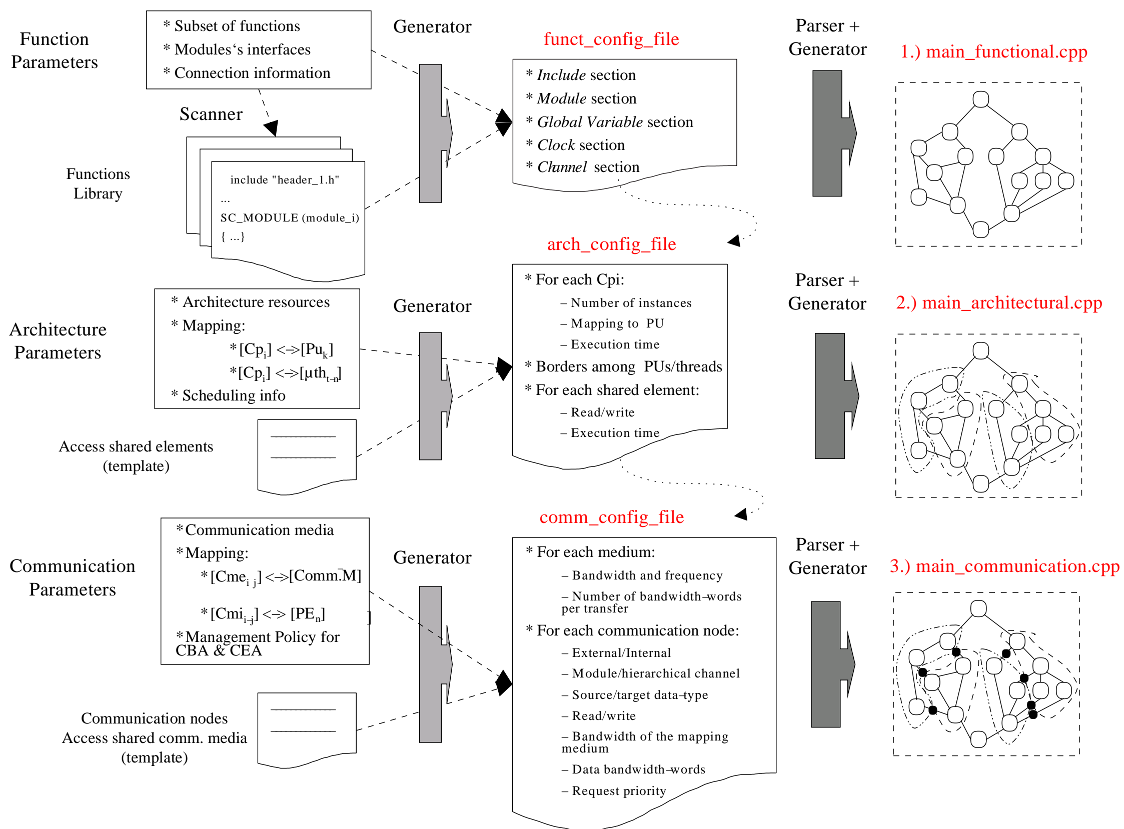

Figure 5.1: ASCSG Implementation Procedure

The input parameters for the creation of each model as well as the format of the corresponding configuration file are explained in the next Sections. As depicted in Chapter 3.5, these input parameters are provided by previous steps in the design flow.

5.2.1 Creation of the Functional Model

When building up the Functional model for the selected application, the subset of required functions is defined. After that, a scanner looks inside the library of functions

to extract the interface information for each SystemC module. Subsequently, a user interface allows the designer to introduce the connections between modules. A port type of and data type of check is then performed in order to avoid a connection error. Taking into account the input information, a functional configuration file is generated which is parsed further on for the creation of the functional executable file.

5.2.2 Creation of the Architectural Model

The structure and the resources of the target architecture have to be provided when pursuing the Architectural model. Therefore, a list is defined which contains the available processing units (PU_k) (either embedded RISC processors (PE_n) with multi-threading capability and no RTOS running on them, or hardware blocks (AC_m) accelerating certain functions), memory units (M_p) and interface units to the external world. Furthermore, for each embedded processor the number of supported threads has to be specified.

The processing time of each *computation node* mapped to the diverse processing units is pre-determined. These execution time values are stored in a library, Table 5.1, which is accessible by the mapping process. The entries marked with an "X" mean that the related mapping is not feasible.

Table 5.1: Possible Mapping Execution Times

	PU1	PU2	...	PUk	...	PUn
Cp_1	$t[Cp_1/PU_1]$	$t[Cp_1/PU_2]$	—	$t[Cp_1/PU_k]$	—	$t[Cp_1/PU_n]$
...	—	—	—	—	—	—
Cp_i	$t[Cp_i/PU_1]$	X	—	$t[Cp_i/PU_k]$	—	$t[Cp_i/PU_n]$
...	—	—	—	—	—	—
Cp_m	$t[Cp_m/PU_1]$	$t[Cp_m/PU_2]$	—	X	—	$t[Cp_m/PU_n]$

The mapping process performs the assignment of *computation nodes* to the available processing units and, in the case of implementation in multi-threading embedded processors, the assignment of *computation nodes* to the threads within a processor. Moreover, the *computation nodes* which make up the initial graph can be instantiated more than once and mapped to different processing units and threads.

The information contained in the architectural configuration file for each *computation node* is: number of instances; mapping of each instance (processing unit number and

priority); and, depending on the mapping, its execution time, which is extracted from the Mapping Execution Times library (Table 5.1).

After the mapping has been performed, the borders between processing units are detected. A *fifo channel* is introduced between two consecutive *computation nodes* which belong to different processing units and are not shared by other nodes. This prevents the loss of information between such nodes. The case of *computation nodes* triggered by diverse nodes running in parallel is solved by providing a mechanism that arbitrates the requests coming from the different nodes attempting to access the shared *computation node* at the same time or while it is busy (see Figure 4.4). In the case of a simultaneous access attempt, the arbitration mechanism takes into account the priority of the *computation nodes* which initiate the access attempts to decide which *computation node* accesses the shared node first. As already mentioned in previous Chapter 4, such a mechanism is referred to as a *shared element* mechanism (Shel$_j$).

In order to model such arbitration mechanism, a template, shel<SHEL_EXEC>, has been own defined. For its customisation, the execution time of the shared *computation node* (SHEL_EXEC) is required. Moreover, an *access node* (Acc$_{i\text{-}j}$) is interleaved between the *computation node* that initiates the access attempt and the shared *computation node*. It is responsible for requesting the read/write access to the Shel$_j$ and waits until it grants access. A template for it has also been defined, acc<T, R_W>, which is customised providing the data type of the *computation node* that initiates the access attempt (T), whether it is a read or write access (R_W) and the priority of the request. This last parameter is passed as a constructor argument to the node module. In the case in which either the transmission initiator or the receiver is implemented as a hierarchical channel (which implements the ch_if<T> interface), the template acc_ch<T, R_W> has to be used instead.

Template 1 Templates for *Shared Elements* and *Access Nodes*

```
template <int SHEL_EXEC>
 class shel: public shel_if, public sc_module {...};
template <class T, bool R_W>
 class acc: public sc_module {...};
template <class T, bool R_W>
 class acc_ch: public sc_module {...};
```

Figure 5.2 below depicts the access for writing and reading of two instances of two *computation nodes* (Cp$_i$[n] and Cp$_i$[n+1] for writing and Cp$_k$[n] and Cp$_k$[n+1] for reading) mapped to two different embedded processors (PE$_n$ and PE$_{n+1}$), to two shared ones (Cp$_j$[n] and Cp$_j$[n+1]) mapped to an accelerator (AC$_m$). For each data segment to be transfered an *access node* (Acc$_{i\text{-}j}$) has to be inserted. It accesses the

corresponding *shared element* mechanism (Shel_j) through a port that is able to call the interface functions implemented in the Shel_j (`sc_port<shel_if>`).

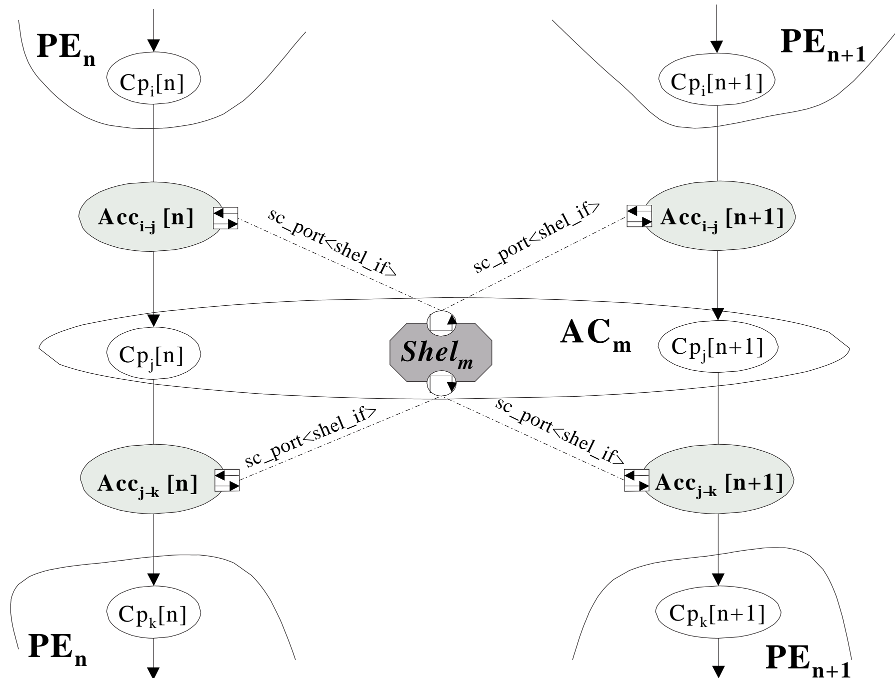

Figure 5.2: Access to a Shared *Computation Node*

5.2.3 Creation of the Communication Model

The last model, the Communication model, includes the characteristics of the Functional and Architectural models and introduces the communication architecture of the system. A list of the available communication media is defined for the external communication. Furthermore, the available threads within each embedded processor, which was already defined for the construction of the Architectural model, is also used here for the definition of the inter-thread synchronisation.

Each time a transfer through a certain communication medium takes place, an *external communication node* ($\text{Cme}_{\text{i-j}}_\text{m}_\text{y}$) is inserted, as shown in Figure 5.3. Such a node provides the access of a processing unit to the selected communication medium. It also calculates the duration for a certain transfer, taking into account the data to be transmitted and the bandwidth and frequency of the communication medium. Each *external communication node* has to be mapped to the medium where the transfer takes place.

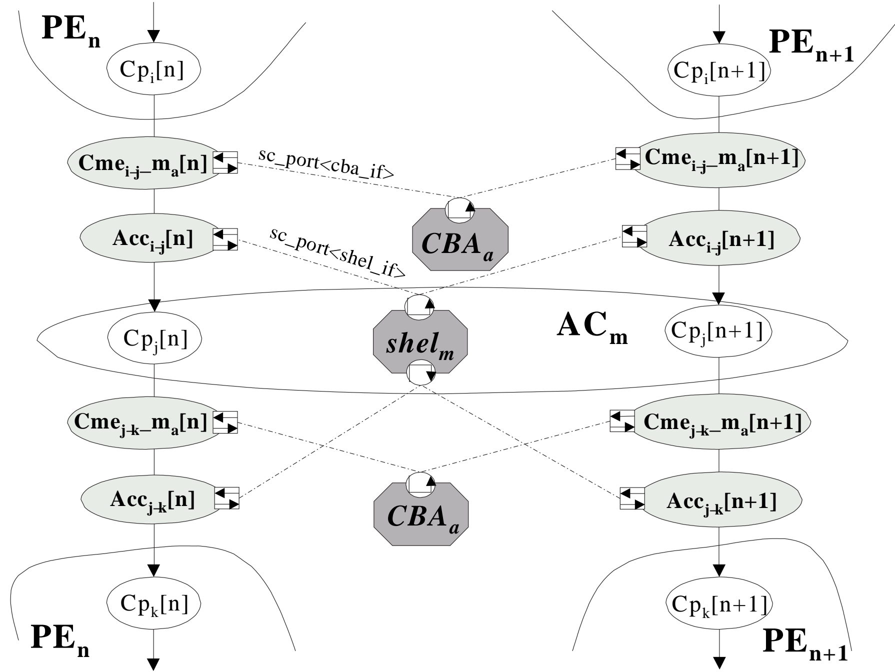

Figure 5.3: Access to a Shared Communication Medium

This mapping information is provided as an input parameter for the creation of the Communication model. For the modelling of such nodes, a template has been defined, cme<T, M_BW, M_LW_T, R_W>, whose customisation requires the following parameters: source/target data type (T); bandwidth of the mapping medium (M_BW); number of words of bandwidth to be transmitted at once through the medium (M_LW_T); whether it is a read or write request (R_W); and the priority of the request. This last parameter is passed as a constructor argument to the node module. In the case in which either the *computation node* which initiates the write transmission or the *computation node* which initiates the read transmission is implemented as a hierarchical channel (which implements the ch_if<T> interface), the template cme_ch<T, M_BW, M_LW_T, R_W> has to be used instead, as seen in Figure 5.4.

Template 2 Template for *External Communication Nodes*

```
template <class T, int M_BW, int M_LW_T, bool R_W>
 class cme: public sc_module {...};
template <class T, int M_BW, int M_LW_T, bool R_W>
 class cme_ch: public channel_if<T>, public sc_module {...};
```

Furthermore, each communication medium has an associated Command Bus Arbiter

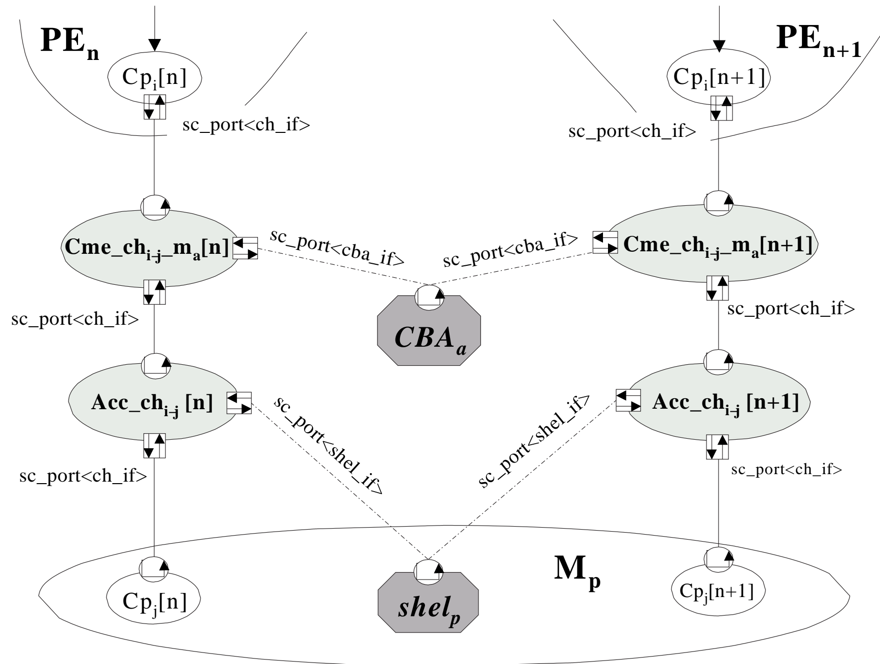

Figure 5.4: Access to a Shared Communication Medium from/to a Hierarchical Channel

(CBA_y), which implements a management policy in order to select the first request to be served. It is accessed by the related *external communication nodes* ($Cme_{i\text{-}j}_m_y$) through a port which is able to call the interface functions implemented in the CBA_y (`sc_port<cba_if>`). A template for it has also been created, `cba<M_BW, M_LW_T, M_FREQ>`. For each communication medium, a CBA_y has to be included which is accessed by the corresponding $Cme_{i\text{-}j}_m_y$. The required parameters for its customisation are the bandwidth (`M_BW`), the number of words of bandwidth to be transmitted at once (`M_LW_T`) and the frequency (`M_FREQ`).

Template 3 Template for Command Bus Arbiters

```
template <int M_BW, int M_LW_T, int M_FREQ>
 class cba: public cba_if, public sc_module {...};
```

Analogously, each time a synchronisation among threads has to be modelled, an *internal communication node* ($Cmi_{i\text{-}j}_n_t$) is inserted, as can be seen in Figure 5.5. The mapping information between the *internal communication nodes* and the corresponding multi-threading embedded processors is also provided as an input parameter for the creation of the Communication model. A template for the modelling of these nodes is

provided, `cmi<T, Ev_Sw>`, whose customisation requires only the source/target data type (`T`), whether the node waits for an event or signalises a context switch (`Ev_Sw`) and the thread number which initiates the request. This last parameter is passed as a constructor argument to the node module. In the case in which either the *computation node* which initiates the write transmission or the *computation node* which initiates the read transmission is implemented as a hierarchical channel (which implements the `ch_if<T>` interface) the template `cmi_ch<T, Ev_Sw>` has to be used instead.

Template 4 Template for *Internal Communication Nodes*

```
template <class T, bool Ev_Sw>
 class cmi: public sc_module {...};
template <class T, bool Ev_Sw>
 class cmi_ch: public channel_if<T>, public sc_module {...};
```

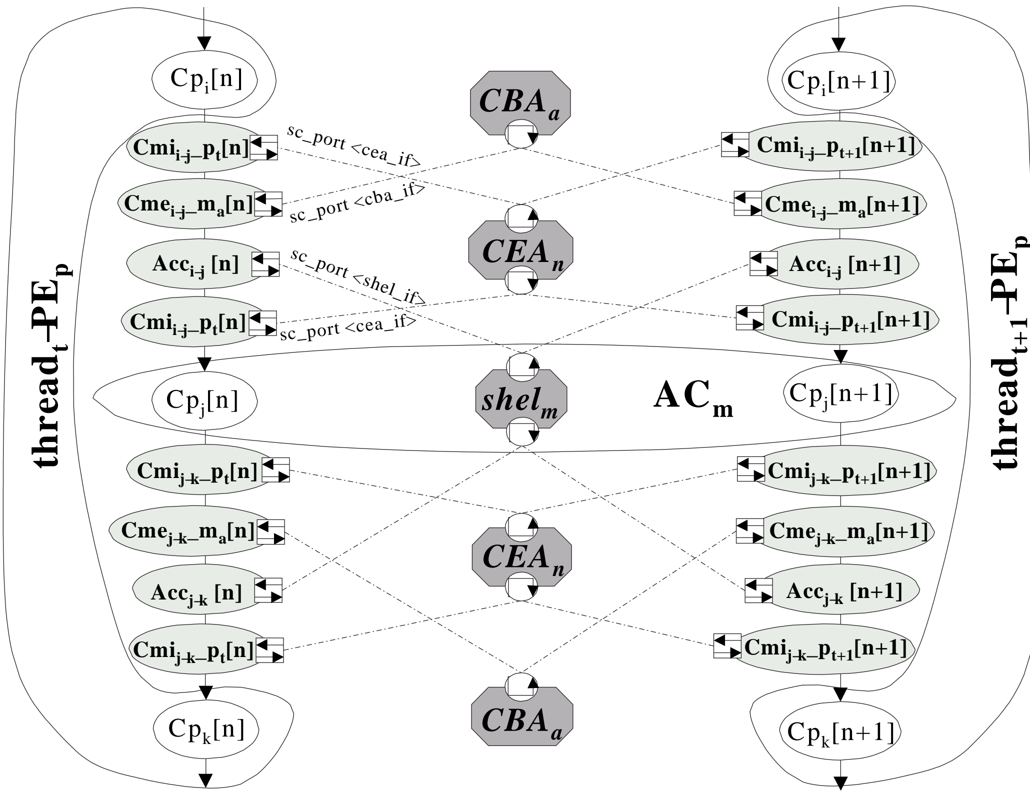

Figure 5.5: Context Switch and Context Event Notification in a Multi-Threading Processor

The synchronisation among threads within a multi-threading embedded processor is arbitrated by the Context Event Arbiter (CEA_n), which is accessed by the *internal*

communication nodes ($Cmi_{i\text{-}j_}n_t$). It implements a management policy in order to decide which is the next thread within the processor to run. Each $Cmi_{i\text{-}j_}n_t$ has to be mapped to the multi-threading embedded processor whose threads it synchronises. For each multi-threading processor, a CEA_n is instantiated, whose customisation requires the number of supported threads (`N_UTH`). A template `cea<N_UTH>` has been defined for this purpose.

Template 5 Template for Context Event Arbiters

```
template <int N_UTH>
 class cea: public cea_if, public sc_module {...};
```

5.3 Automation Approach

Once the input functional specification and the target architecture, the building blocks and the communication structure, have been defined, the exploration of several mapping–scheduling alternatives requires only the information concerning the mapping of the *computation nodes* onto the processing units, the scheduling information and the mapping of the *external communication nodes* onto the communication media and the mapping of the *internal communication nodes* onto the multi-threading embedded processors. With these parameters, the implementation procedure presented in Section 5.2 is automated and the Communication ASCSG for every mapping–scheduling alternative is extracted. The evaluation and comparison of the corresponding output graphs give the designer the required references for choosing the most adequate alternative.

For the development of the scanners and parsers required by the automation process, the Lex and Yacc tools ([101]) have been used (in particular, the GNU free version of Lex, *Flex* ([102]), and of Yacc, *Bison* ([103])). Lex and Yacc help by generating programs that transform structured input. Furthermore, they allow for rapid application prototyping, easy modification and simple maintenance of programs. Specifically, Flex is a tool for generating scanners, i.e., programs which recognise lexical patterns in text. And Bison is a general purpose parser generator that converts a grammar description for a context-free grammar into a C program to parse that grammar.

5.3.1 Data Structure for the Implementation

A common design approach is followed with the data structures shown in Figures 5.6, 5.7 and 5.8, which are further annotated in subsequent steps. These sample data structures are intended to clarify the semantics of the automation.

For each *computation node* (Cp_i) a data structure with the following fields, as shown in Figure 5.6, is built up: name (CP_NAME); identification (ID); whether it is modelled as a hierarchical channel or as a module (HC_MOD); and number of instances (NUM_INST). Furthermore, the data structure comprises a pointer to the contained ports (`port_list`); and a second pointer to the mapping information of each instance (`mapp_list`).

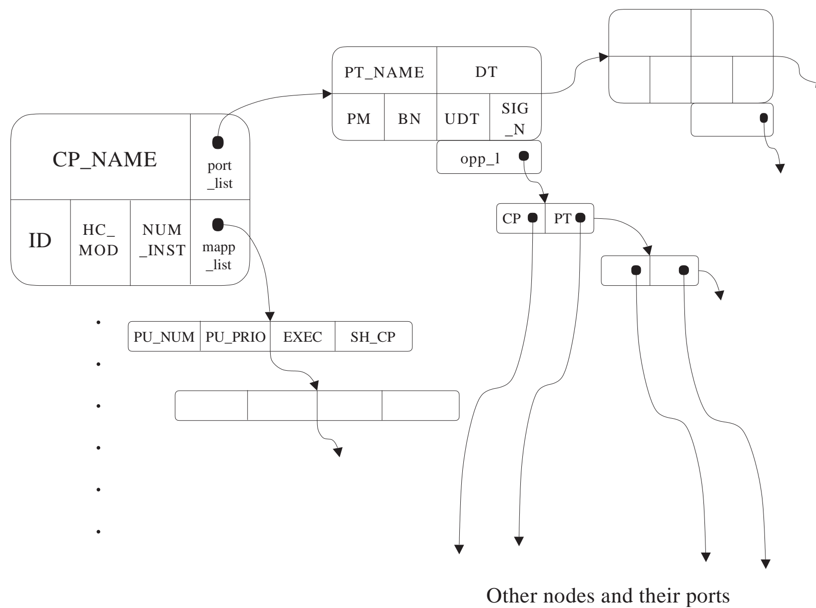

Figure 5.6: *Computation Node* Data Structure

In the same way, the required information for each port is: name (PT_NAME); data type (DT); whether it is a SystemC/C++ built-in or a user-defined data type (UDT); port mode (PM); whether it is bound (BN) and the name of the signal attached to it (SIG_N). Moreover, a pointer, `opp-l`, to the attached *computation node* (CP) and the related port (PT) is also added.

The mapping process associates to each instance the parameters concerning the processing unit number (PU_NUM), the priority of the processing unit (PU_PRIO), its execution time (EXEC), and whether it is a shared node (SH_CP).

As has been seen in the previous Section, the detection of a *computation node* triggered by multiple *computation nodes* implies the insertion of an *access node* in between and an arbitration mechanism, which is referred to as a *shared element* mechanism. The data structures for these two elements are shown in Figure 5.7.

Each *access node* contains its name (ACC_NAME), identification (ID), data type to pass across (DT), and whether it attempts a read or a write request to/from the shared

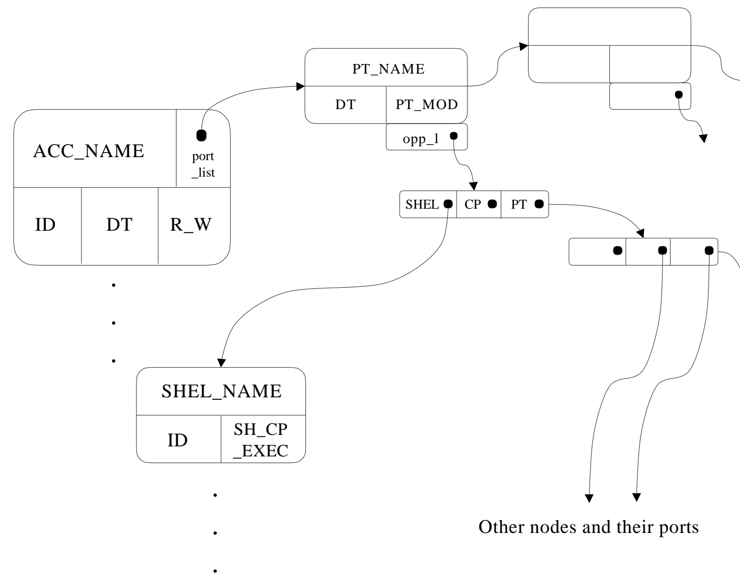

Figure 5.7: *Access Node / Shared Element* Data Structure

computation node (R_W). A pointer to the list of its ports, `port_list`, gives access to the information related to each port: Name (PT_NAME); data type (DT); and port mode (PT_MOD). A further pointer, `opp-l`, binds each port either to the *computation node* (CP) concerned and the related port (PT), or to the arbitration mechanism, the *shared element* (SHEL).

The information required for each *shared element* in order to further customise the related template is: name (SHEL_NAME); identification (ID); and the execution time of the shared *computation node* (SH_CP_EXEC).

The last step towards the creation of the communication ASCSG comprises the insertion of *internal and external communication nodes* for the synchronisation among threads and the access to the communication media, respectively. The customisation of such additional nodes requires certain parameters which are further annotated in their data structure as shown in Figure 5.8.

Each *communication node* data structure comprises its name (CM_NAME), identification (ID), type (internal or external) (TYP), whether it is modelled as a hierarchical channel or as a module (HC_MOD), whether it attempts a read or a write request (R_W) in case of an *external communication node* or whether it waits for an event or signalises a context switch in case of an internal one, bandwidth of the communication

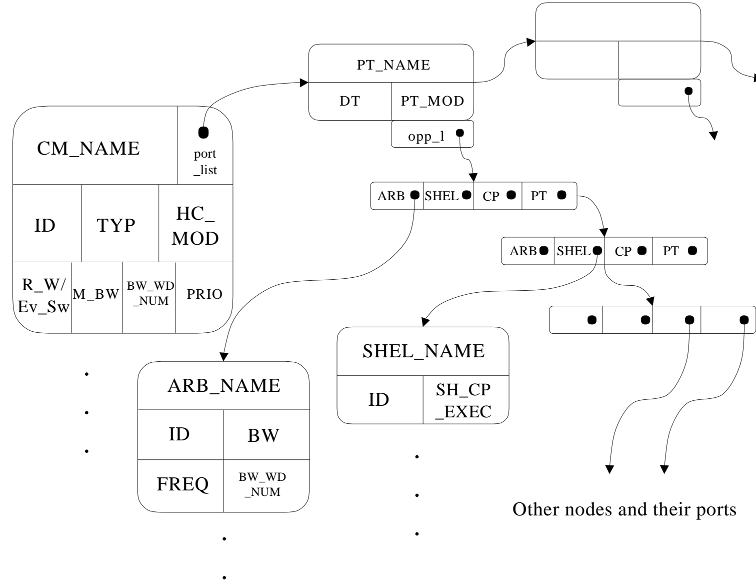

Figure 5.8: *Communication Node* / Arbiter Data Structure

medium (M_BW)[1], number of words of bandwidth to be transmitted at once through the medium (BW_WD_NUM)[2], and priority of the initiator (PRIO). A pointer to the list of its ports, `port_list`, gives access to the information related to each port: name (PT_NAME); data type (DT); and port mode (PT_MOD). A further pointer, `opp-l`, binds each port either to the *computation node* (CP) concerned and the related port (PT), or to the arbitration mechanism, the *shared element* (SHEL), or to the arbitration mechanism (ARB), the Context Event Arbiter and the Command Bus Arbiter for the *internal and external communication nodes*, respectively.

The information required for each arbitration mechanism in order to further customise the related template is: name (ARB_NAME); identification (ID); bandwidth (BW)[2] and frequency (FREQ)[2] of the communication medium; and number of words of bandwidth to be transmitted at once through the medium (BW_WD_NUM)[2].

5.3.2 Configuration Files

The Functional model is created using a subset of the library of functions and the explicity interconnection information of their ports, which is completely specified in the functional configuration file.

[1]Only required in case of an external *communication node*

Since the task of specifying the interconnects is bit consumed, an option of interactively creating the initial configuration file is implemented to which the subset of functions is the input. The input files are scanned for the port definition, types, names and only the feasible links are displayed from which the user can select the appropriate one. The configuration file for the Functional model can also be tuned manually, thus reserving time for its creation. This configuration file is then parsed and fed to the data structure defined above in Figure 5.6 (with the exception of the elements of the `mapping list`, which are defined after the mapping process). Finally, these linked lists are simply traversed and processed iteratively in order to generate the simulatable Functional model (`main_functional.cpp`).

Specifically, the simulatable Functional model is a SystemC main file which is divided into the following sections: include files; channel declaration; module instantiation; and trace file section. Thus, the functional configuration file consists of the following sections:

- Include section: Contains the name of the header files to be included;
- Module section: Comprises the name of the modules along with their parameters and the list of all their ports and ports details. Moreover, it contains the list of modules and ports to which each of the own ports are connected;
- Global Variable section: Includes more general information as, for example, the desired name of the output file, the trace file name and its type, i.e., vcd or wif;
- Clock section: Specifies the details for every clock and their parameters;
- Channel section: Defines the links between ports, either through SystemC hardware signals or through SystemC primitive channels.

The steps towards creating the Architectural model imply the mapping of every instance of each *computation node* to the selected functional unit of the target architecture. This is done by making the *computation node* hold more information inside its data structure. This information includes, for each instance, the processing unit number to which it is mapped, its priority and the corresponding execution time, which is picked up from the Mapping Execution Times library (see Table 5.1).

With these parameters a process detects whether shared *computation nodes* exist and, in the affirmative case, the corresponding flag inside its data structure is set. Furthermore, a list of required *access nodes* and another of the corresponding *shared elements* are created. The required information for their customisation is extracted from both the access initiator and the shared *computation node*.

In practice, the architectural configuration file extends the functional configuration file that has to be available in this step. The Module section additionally comprises the mapping information of each *computation node* instance and the lists of *access nodes* and *shared elements* along with their details. This architectural configuration file is then parsed and fed to the data structures defined in Figures 5.6 and 5.7. Finally, these linked lists are simply traversed and processed iteratively in order to generate the simulatable Architectural model (`main_architectural.cpp`).

The last step involved in creating the Communication model carries out the insertion of *internal and external communication nodes* and the corresponding arbitration mechanisms to access the multi-threading embedded processors and shared media, respectively. For these additional elements, two new linked lists are created. The mapping of the internal and external transmissions to the target embedded processors and communication media gives the input information required by their customisation.

The communication configuration file extends the Module section of the architectural configuration file with these two new linked lists of *communication nodes* and arbitration mechanisms. This configuration file is again parsed and fed to the data structures defined in Figures 5.6, 5.7 and 5.8. Once again and lastly, these linked lists are simply traversed and processed iteratively in order to generate the simulatable Communication model (`main_communication.cpp`).

In summary, it has been shown that by means of successive configuration files, which are built one on top of the previous one, the executable SystemC files for the three models which make up the ASCSG can be generated. The configuration file for the Functional model is defined once at the beginning and remains unchanged during the exploration of different architecture–partition alternatives. As depicted in Figure 5.9, each time a new alternative is to be tested, it is only necessary to add the information concerning the architecture–partition configuration of this new alternative. With this configuration information, the corresponding configuration files are created. These are then parsed and the Architectural and Communication ASCSG are generated. The evaluation by means of simulation of the last model, the Communication ASCSG, delivers the pursued performance estimation values for the alternative under study.

As already introduced in Chapter 1, the proposed performance estimation methodology is thought to be integrated in a hardware–software co-design procedure, where several target architectures have to be explored and an easy re-mapping of the functions onto the target architecture should be possible. The hardware–software co-design procedure defines the input parameters required by the proposed methodology and, therefore, it is only necessary to agree on the format to pass these parameters to the corresponding configuration files previously presented.

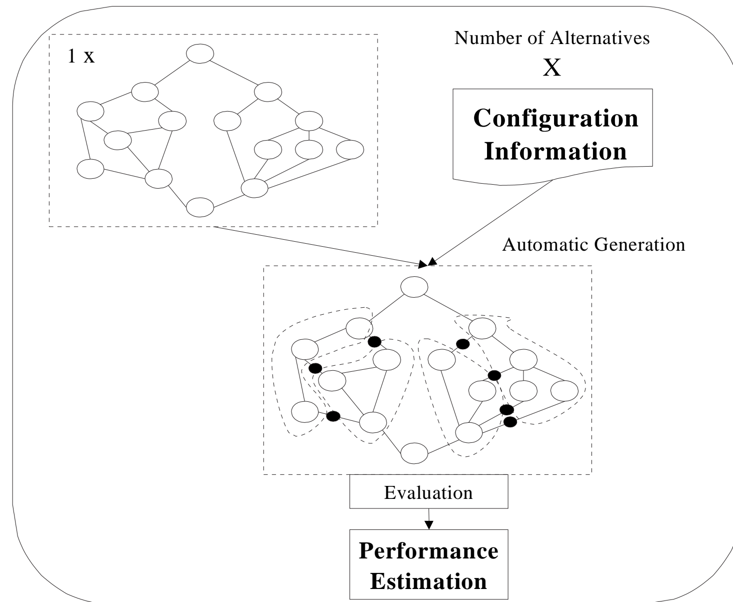

Figure 5.9: Performance Estimation by Means of Configuration Information

The procedure towards the exploration of architecture–partition alternatives can be automated in this way. This automation, together with the implementation of an exploration strategy, will facilitate the analysis of several possible alternatives. It will lead, further on, to the selection of the most suitable solution that meets the design requirements in terms of performance. Nevertheless, the solution reached at the end of the exploration process depends on the strategy how the performance data is used for changing the solution.

Chapter 6

Methodology Verification

6.1 Chapter Introduction

In order to position the advantages and disadvantages of the proposed methodology, its results in terms of evaluation time and accuracy have to be verified. For this purpose, the results achieved by the method based on the ASCSG have to be compared with the results delivered by a model at a lower level of abstraction. A cycle-accurate model is chosen as a reference for the verification of the proposed methodology.

In order to perform this comparison, a case study is selected for which both procedures, the ASCSG and the cycle-accurate model, are applied. The simulation time and output throughput delivered by the cycle-accurate model are compared with the values provided by the methodology based on the ASCSG. The same procedure is repeated for diverse architecture–partition configurations in order to cover different possible scenarios. A flexible hardware–software co-simulation platform substantially facilitates this step. It consists of several modules which simulate hardware specific blocks, software running onto multi-threading embedded RISC processors and their interaction with memory blocks and interfaces with the external world.

The system level language SystemC has been used as the specification language for both the proposed system level performance estimation methodology and the modelling of the cycle-accurate co-simulation platform. This issue makes the comparison of results of both models easier and more reliable.

This Chapter is organised as follows: First of all, the existing co-simulation techniques are explained in order to show the different possibilities when building a co-simulation platform. Then, the selected target platform architecture is presented, followed by the modelling of the co-simulation platform. Lastly, the implementation

of the skeleton of the co-simulation platform together with its different functional and memory units is depicted.

6.2 Hardware–Software Co-Simulation Techniques

A hardware–software co-simulation technique is used to verify the correct interaction of hardware and software. The available techniques for hardware–software co-simulation can be characterised by a number of factors ([4]), which include:

- Performance: The co-simulation techniques offer different levels of operational capacity. It is, however, the combined hardware–software performance of the system that is important. For instance, it is irrelevant how fast the software side of the system is executed if the hardware simulation is left unaccelerated. When dealing with performance, it is crucial to address the acceleration of the entire system rather than that of the separate hardware or software components;
- Timing accuracy: The accuracy of the results must be maintained when increasing the level of abstraction of the model in order to achieve equitable values. No level of performance can be justified if it cannot correctly represent the expected behaviour of the system under test. Optimisations that cause a reduction in the level of accuracy must be under user control and discretion. For example, many interactions between hardware and software are sensitive to timing constraints. The absence of accurate instruction timing may make it impossible to truly validate the hardware–software interface;
- Model availability: In some co-simulation techniques, a Bus Functional Model (BFM) of the processor is required in order to model the transactions on the bus. Nevertheless, if a detailed simulation of the processor is required, an Instruction Set (IS) Model is needed for meaningful hardware–software co-simulation. Furthermore, the performance of the interface connecting such models and the rest of the system must be highly optimised;
- Visibility of internal state for debugging purposes: A common debugging environment must offer a layered view of the software at both the process and code levels and the hardware depending on the user needs. Visibility of the processor states and ability to create breakpoints are important elements of an effective hybrid system debugging environment;
- Cross-domain optimisation: There are also many cross-domain optimisations that can be made. In essence, it relies on the ability to selectively suppress activity

from crossing the hardware–software boundary in a manner that does not result in a loss of accuracy or accessibility to data.

The existing co-simulation techniques to date can be classified into techniques requiring processor models and techniques not requiring processor models, as presented in [104].

6.2.1 Techniques Requiring Processor Models

These techniques are used when a full model of the processor used in the co-simulation is available. All of the important elements for state storage of the processor are preserved but the data path that connects them is abstracted out of the model. The processor is described in terms of its instruction set. That is, a collection of instructions is modelled, where each instruction defines a relationship between constituents that are internal (registers, on-chip memory) or external (on-board memories) to the processor. Full bit level accuracy exists within the models.

Instruction set models allow both high-level and assembly code to be executed and debugged. They can exist at three different levels of accuracy:

- Nano Second Accurate Timing model: The most accurate software model uses a processor model that has a nano second accurate timing for all the pins and the complete functionality. However, the simulation is slow. The propagation of many events is necessary because each pin can change at unique times. Typical performances for these types of models are in the 1 to 100 instructions per second range;
- Cycle-Accurate model: Cycle-accurate models cover the internal and external state of the model at every bus cycle. This means that the exact bus behaviour of the target device will be observed. The transitions occur at each clock edge. This means that there are fewer unique event times in the system, making the simulator run faster. Typical performance numbers for this type of model are 50 to 1,000 instructions per second;
- Instruction Set Simulator (ISS): This model of the processor emulates the instruction set accurately, which means that the values in registers and memory are correctly modelled ([105]). While they do guarantee that all of the correct bus cycles will be performed and that the total number of cycles for the instruction will be correct, they do not guarantee that the bus operations will occur during the correct clock cycles. They also do not attempt to assure the correct internal state at individual clock boundaries. These models are adequate for most

situations. These kinds of instruction emulation models can run from 2,000 to 20,000 instructions per second. Some inaccuracy may occur when bus sharing for contention exists, since the exact time of the bus cycles may not be correct.

6.2.2 Techniques not Requiring Processor Models

If software and hardware communicate through communication methods such that the time between communications has no effect on functionality, a faster method of simulation is possible. At this level, there is no need for a processor model. The software is compiled on the host machine and linked with the simulator. The techniques used can be divided into synchronised handshake, virtual hardware, bus functional models and other hardware modellers.

- Synchronised Handshake: The detailed communication between hardware and software is then replaced with a synchronising handshake. The software runs at the speed of the workstation and the hardware is simulated on the hardware simulator. The overall simulation speed will be dominated by the hardware simulator performance and can be measured in MIPS (Million Instruction Per Second);
- Virtual Hardware: Only software is simulated creating for it a virtual operating system and the corresponding machine code in pure software that has no relation to the real hardware. Disk I/O is mapped into the host operating system. It is one of the fastest simulations but with the least accuracy;
- Bus Functional Model (BFM): Bus functional models allow the simulation of the hardware but there is no simulation of the software ([105]). The performance is limited by the hardware simulation. A bus functional model of a processor encapsulates the bus functionality of a processor. Such a model can only execute bus transactions on the processor bus (with cycle accuracy) but cannot execute any instructions. The BFM provides a programming interface that can be used by the software directly. Since the software runs on the host processor (on which development is done), this model is un-timed because the software execution time is not accurate. A BFM is therefore an abstract processor model that can be used to verify how a processor interacts with its peripherals. Thus, a BFM represents a key component in any co-verification solution. In the design methodology used for the implementation of the current hardware–software platform, a BFM is used throughout the design process;
- Hardware Modeller: Systems based around complex processors often use hardware modellers to emulate the CPU. Current implementations of hardware modellers have the most accurate models from a functionality point of view but have

only modest performance. Hardware modellers typically run in the 10 to 50 instructions per second range;

- Emulation: Emulators map the hardware part of the design down onto programmable hardware. The programmable hardware implements the behaviour of the loaded design and realises its execution at the supported frequency (e.g. 1 MHz to 10 MHz). Historically, these devices have suffered from high cost, very long design iteration times, and a lack of comprehensive debug capabilities. Emulation provides the closest simulation to a real prototype that is possible.

In the present work, the decision concerning the model to be used as reference is taken as a compromise between modelling effort and content of system details. Whereas a real implementation and even the design of a model at logic or RT level of a complex system would require a big design effort and would take a lot of time, a cycle-accurate model comprises more details of the system than the proposed methodology and delivers accurate cycle counts. That is the reason for choosing a cycle-accurate model as a reference for the verification of the proposed methodology.

6.3 Target Platform Architecture

As a basis for the development of the co-simulation platform, the target architecture depicted in Figure 6.1 has been chosen. It consists of a bus-based System-on-Chip (SoC) architecture which comprises diverse processing units (PU_k), memory units (M_p) and the required interface units to the external world. The processing units can either be embedded RISC processors (PE_n) with multi-threading capability and no RTOS running on them, or hardware blocks (AC_m) accelerating certain functions. The memory units can be either global or local to specific blocks. The communication structure consists of two buses, a command and a data bus, which are accessed by every unit, except for the access to the local memories where a point-to-point connection is implemented. When more than one master attempts to initiate an access to one of the common buses, an arbiter decides which request is served first.

As already mentioned in Chapter 1, a case study is selected for the comparison of the results delivered by both procedures, the ASCSG and the cycle-accurate model. The selected case study deals with the processing of packets at the edge of the network, where high speed in the data path is required. The functions to be implemented are restricted to the user plane functions (see Chapter 2.4) from the input side of a packet processor (i.e., a router or a gateway). Therefore, the *TCP/IP processing +*

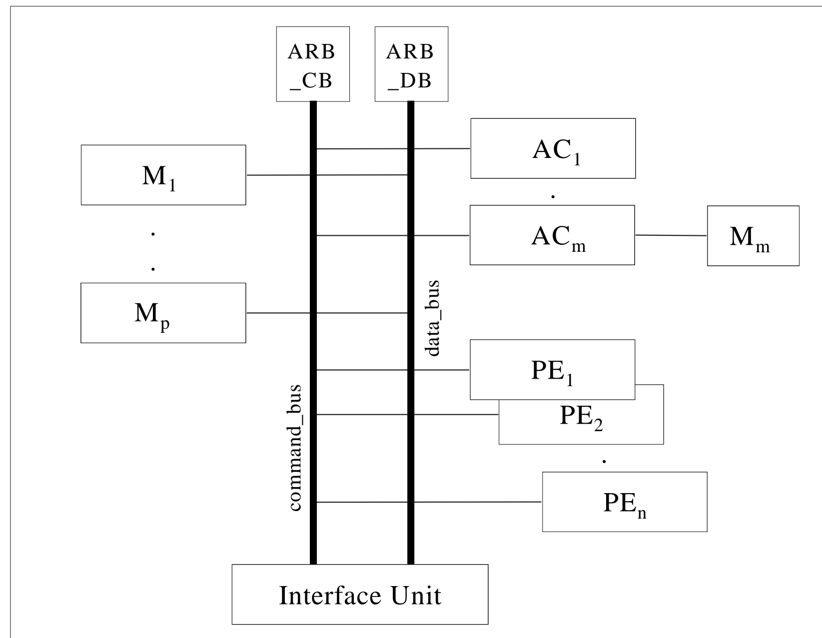

Figure 6.1: Target Platform Architecture

DiffServ functionalities are supported. It comprises, for example, packet recognition, packet verification, checksum computation, mark and modification, packet classification, policing and the implementation of the longest prefix match algorithm amongst others. The target platform architecture depicted in Figure 6.1 is conceived to cope with such functionalities. The structure of this architecture is based on the Network Processor from Intel IXP1200 ([106]), which is mainly intended to process packets at the edge of the network.

6.4 Modelling Approach

Once the existing co-simulation techniques have been examined and the target platform architecture introduced, a decision concerning the modelling of the co-simulation platform for verifying the proposed methodology has been taken. Two different co-simulation techniques have been applied:

- First, a functional model of the platform has been developed. It consists of an Un-Timed Co-Simulation Model which verifies the correctness of hardware and

software working together. For this purpose a Bus Functional Model (BFM) of the processor which encapsulates its bus functionality has been built up.

- Secondly, a Cycle-Accurate Co-Simulation Model has been developed. An Instruction Set Simulator (ISS) of the multi-threading embedded processor has been modelled. It provides accurate cycle counts required by the cycle-accurate performance analysis. The functions of the system specification to be implemented in software have then been translated into machine code which is read by the ISS. A cycle-accurate model of the hardware blocks is also required at this point.

6.4.1 Skeleton of the Co-Simulation Platform

As mentioned in the previous Section, the internal structure of the selected platform architecture is based on the Network Processor from Intel IXP1200 ([106]). It is a fixed architecture which includes six programmable packet engines plus an additional StrongArm processor. The IXP1200 can connect to an external host processor through a 32-bit PCI bus. It also connects directly to up to 256 MB of external SDRAM for storing packet data and up to 8 MB of SRAM to store routing information. It includes a hardware hash unit for generation of polynomial keys, useful for quickly looking up IP addresses. The IXP1200 supports only a single external bus (the IX bus) for packet data. Intel rates the 200 MHz IXP1200 at 3.0 million packets per second (Mpps) when performing Layer 3 routing on 64-byte packets.

The co-simulation platform built up in the present work is a generic model of the IXP1200. It consists of a flexible hardware–software bus-based co-simulation platform able to simulate and verify complex architectures that involve embedded processors and hardware blocks. The number of such internal processing units is made configurable in order to be able to test different architecture–partition alternatives.

As is the case for the IXP1200, the data/command transmission among resources is restricted to a bus-based communication, except for the point-to-point link to access the local memories.

Figure 6.2 depicts the implemented co-simulation platform. The number of multi-threading embedded processors (PE_n) and hardware blocks acting as accelerators of certain costly functions (AC_m) is configurable. Each unit has an identification number that is also used as a priority number when solving concurrent access to a shared communication medium.

The embedded processors work in parallel controlling the whole processing of the packets entering the system. First of all, they pick up the packets from the receive FIFO

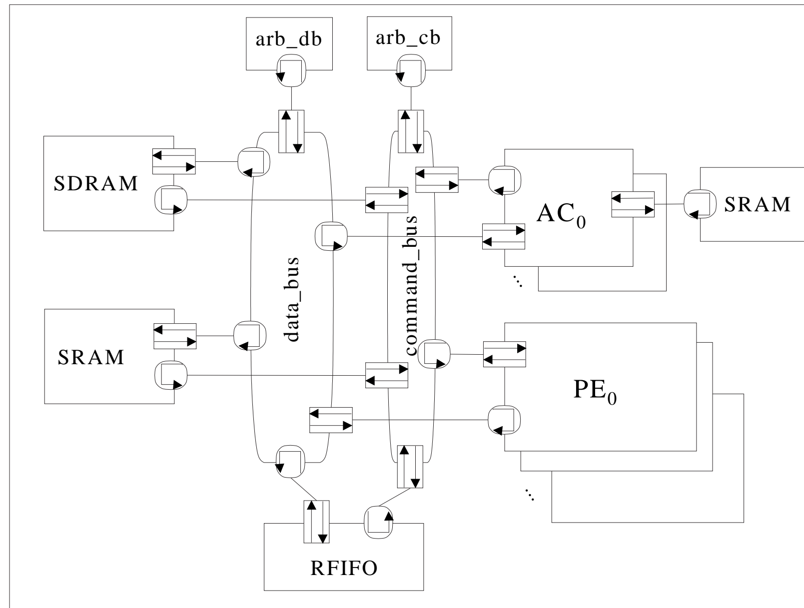

Figure 6.2: Bus-Based Co-Simulation Platform

(RFIFO), where they are stored in the arrival sequence. Secondly, the processors start performing the sequence of functions. Some of them are performed internally, requiring only some table lookups to the global SRAM. Other costly functions are carried out in a dedicated external accelerator and, therefore, the required information has to be passed to the hardware block. Once the processing of a packet has finished, the processor stores it in the packet memory (SDRAM).

Inside each multi-threading embedded processor, a configurable number of contexts [1] is available. A control mechanism cares for the context switches and implements a policy mechanism to decide which is the next thread to run. The instruction memory inside each processor is also divided into as many parts as there are contexts. The corresponding number of program counters point to the different context sections into which the instruction memory is divided.

The embedded processors act as masters when sending a command request to the shared functional units through the command bus. The target functional units store the requests in an input command FIFO and process them in the sequence in which they

[1] The separate programs that share execution time on a processor are referred to as program contexts or contexts.

arrive. The data bus is shared by the functional units to perform the data transmissions indicated in the processed requests. Once a certain command has been completed, the target unit signalises it to the initiator (the master who sent the command request first) by sending it the corresponding event. In a multi-threading embedded processor, such events are the inputs that wake up contexts, which were put to sleep after sending the related requests.

Next, the different functional units included in the platform and its communication structure are explained in detail.

6.4.2 Functional Units

The current co-simulation platform contains the following functional units: embedded RISC processors; hardware blocks acting as accelerators of costly functions; and shared memories.

6.4.2.1 Embedded RISC Processor

The internal architecture of each embedded RISC processor is shown in Figure 6.3. It follows the scheme of the Intel IXP1200. It supports a 32-bit RISC instruction set tailored to networking and communication applications. The frequency of operation is parametrisable and all instructions are executed in a single cycle.

The main features of each embedded RISC processor are: hardware multi-threading support for a predefined number of threads; a programmable instruction memory; two blocks of 32-bit general purpose registers; and two blocks of 32-bit transfer registers for transferring data into and out of the processor; an Arithmetic Logic Unit (ALU;) and a shifter. The size of the instruction memory as well as the size of the general purpose and transfer registers are parametrisable.

Each embedded processor contains a programmable instruction memory that holds the microcode program. The predefined number of threads associated with each processor share the instruction memory. It is 32-bit wide and its size is parametrisable.

Furthermore, each embedded processor supports two blocks of a parametrisable number of 32-bit general purpose registers (A_GPR and B_GPR). They are addressed using absolute addressing, i.e., every register is shared among all the threads within a processor.

Data is moved into and out of the embedded processors via the transfer registers. Each processor supports a parametrisable number of 32-bit bus transfer registers, di-

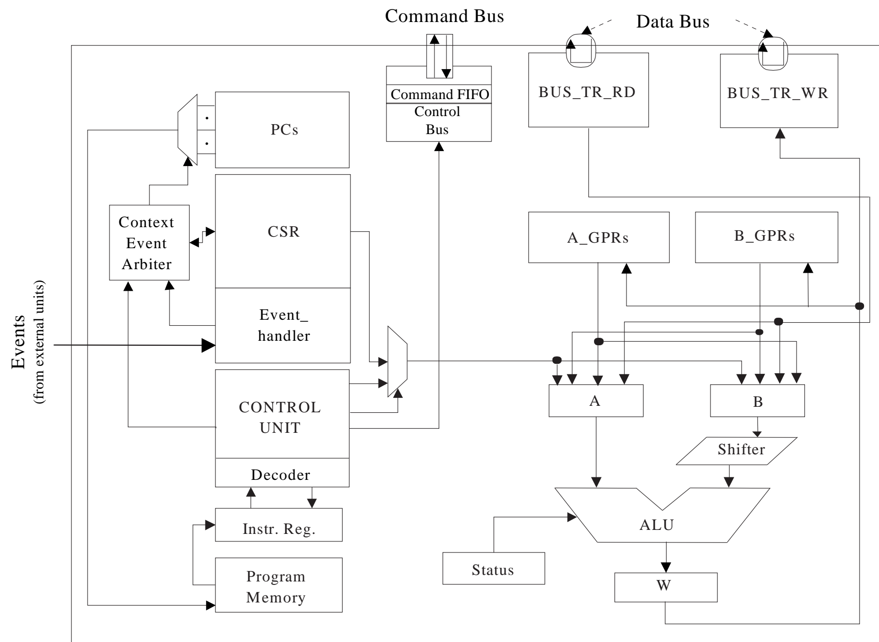

Figure 6.3: Embedded RISC Processor

vided into two sets of read (BUS_TR_RD) and write (BUS_TR_WR) transfer registers. Each register subset connects to the other functional units via a common data bus. They are addressed using absolute addressing.

The embedded processors contain a 32-bit ALU and a Shifter that are capable of performing a single cycle operation. The two inputs of the ALU can operate on data supplied by the read transfer registers, general purpose registers and immediate data within the instruction. The ALU can perform addition, subtraction and logical operations. They generate sign, zero and carry out condition codes which are stored in the Status register.

The embedded RISC processors issue references to the other functional units within the platform via a shared command bus. When a processor thread executes a reference instruction, a command is generated by the Control Bus unit and placed into a two entry Command FIFO within the processor. The access to the command bus is then arbitrated by the Command Bus Arbiter, which determines the processor allowed to access the shared command bus.

Finally, each embedded processor contains a set of control and status registers (CSRs). A processor can access its own set of local CSRs, but only for reading.

Each embedded processor instruction is pipelined through a five stage Control Unit.

Once the execution pipeline is filled with instructions, an instruction is executed in every cycle. Instructions such as branch, jump/return and context switching result in a branch decision that may introduce an aborted instruction to the pipeline that reduces the efficiency of the processor.

Hardware multi-threading support allows the predefined number of program contexts to share execution time on an embedded processor. When a thread is not executing, each program context is preserved in hardware through separate program counters, signal event states, and a register set for each context within the Control and Status Registers (CSRs). When a thread is put to sleep, a context switch occurs and another program context begins executing. The overhead associated with switching contexts is a maximum of one instruction cycle.

Figure 6.4 depicts the hardware blocks which implement the multi-threading support. The Events from the other functional units are held and then handled by the Event Handler module, which sets the corresponding flags within the related registers of the CSR. Furthermore, there is a Context Event Arbiter that determines the processor thread to be allowed to run when a thread is put to sleep.

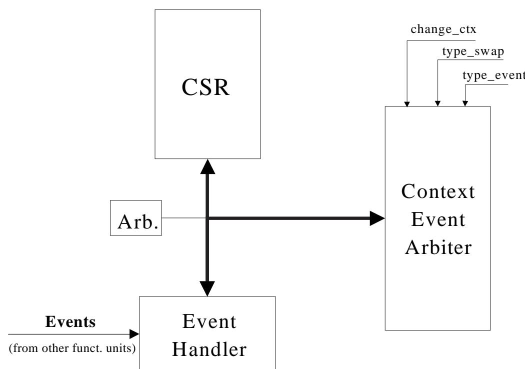

Figure 6.4: Multi-Threading Modelling

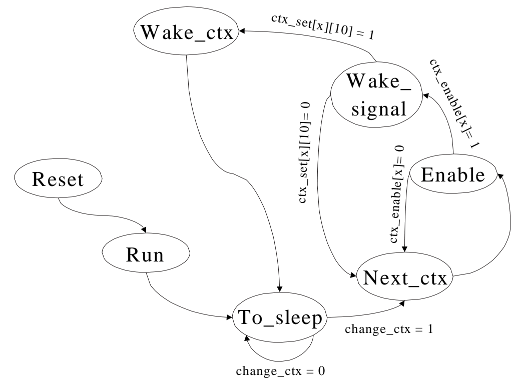

Figure 6.5: FSM of Context Event Arbiter

Once a context switch command arrives, the Context Event Arbiter makes the decision based on the values of the corresponding bits within the concerning registers of the CSR. The transitions between states is shown in Figure 6.5. The Finite State Machine (FSM) starts in the *Reset* state, where the FSM is initialised. It is immediately followed by the *Run* state, where the contexts are activated. The information related to each context is stored in the corresponding registers within the CSR during the *To_sleep* state. The FSM remains in this state until a context switch command arrives from the control unit. Once it occurs, the *Next_ctx* state chooses the next context to run. In the following two states, *Enable* and *Wake_signal*, it is checked whether the

chosen context is enabled and ready to run. In the affirmative case the *Wake_ctx* state wakes up the selected context and, in the case in which such context is neither enabled nor ready, a next context is chosen, going back to the *Next_ctx* state.

6.4.2.2 Hardware Blocks – Accelerators

The hardware units are thought to perform certain costly functions such as packet classification and the longest prefix match algorithm within networking applications. They communicate with the embedded processors through the command and data bus.

The embedded processors send command requests to the hardware unit which are stored in an internal input Command FIFO. The requests are then processed in the order in which they arrived. The processing of a request implies data transmission to/from the read/write transfer registers of the initiator processor through the data bus. The data required as input by the implemented function is stored in an internal Receive FIFO which is served in the arrival sequence. On the other hand, the pursued results are stored in an internal Transmit Memory in the location reserved for the thread processor which sent the corresponding request. In this way the processing of a read command will pick up the result from the corresponding address in the Transmit Memory and send it to the Read Transfer Register of the initiator processor. In the case in which the result required by a read command is not available at the moment the command is to be processed, the request is enqueued at the end of the input Command FIFO. Once a certain command has been completed, the hardware unit signalises this event to the initiator thread processor, sending the corresponding event to the processor. An internal Control Unit manages the complete procedure and stores the control information of the actual request being processed.

Figure 6.6 shows a generic internal architecture of a hardware block.

The hardware units can own a local memory which is accessed by a point-to-point connection. These local memories are usually dedicated for storing auxiliary tables required by the implemented algorithm. A considerable number of lookups to these tables has to be performed. These usually constitute the bottlenecks.

6.4.2.3 Shared Memories

The shared memories of the platform are the Receive FIFO (RFIFO), the SRAM and the SDRAM. The first one, the RFIFO, stores the arriving packets from the medium access layer. The SRAM is destined to store auxiliary tables and descriptors required by diverse algorithms that will be implemented inside the embedded processors. And, lastly, the SDRAM stores the packet data inside the platform. These functional units

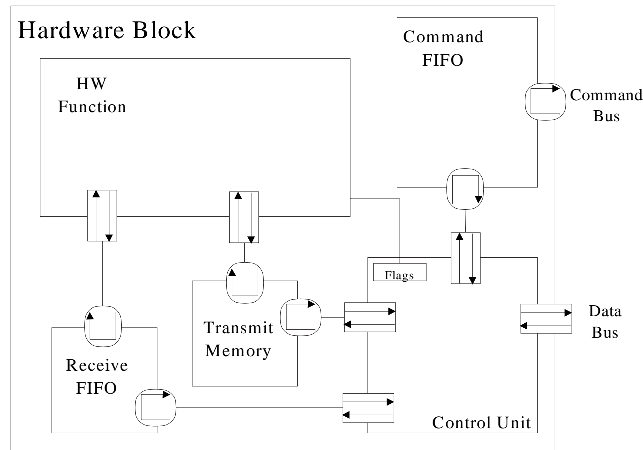

Figure 6.6: Internal Architecture of a Hardware Block

are shared by the different embedded processors and are accessed in the same way as the hardware blocks.

The command requests sent by the processors are stored in an internal input Command FIFO and served in the order of arrival. Once a read/write request is processed, the corresponding data transmission to/from the read/write transfer registers of the initiator processor is performed through the data bus. Once a certain command has been completed, the memory unit signalises it to the initiator thread processor, sending it the corresponding event. An internal Control Unit manages the complete procedure. The internal architecture of a shared memory is illustrated in Figure 6.7.

6.4.3 Communication Structure

The command and data bus constitute the central communication media of the bus-based SoC architecture. Each bus collect the read/write requests from the corresponding units acting as masters of each bus and handle them depending on their priority. In order to perform this function, the buses need the information concerning the data to be transmitted, the priority of the initiator, the source and target of the transmission and, in case it is required, the source/destination address.

During the transmission, the bus status is set to `BUS_WAIT` and, when the transfer ends, the bus status is set back to `BUS_OK`. The bus status only gets the state `BUS_ERROR` when a transfer has failed. An internal routine then determines the cause of the failure.

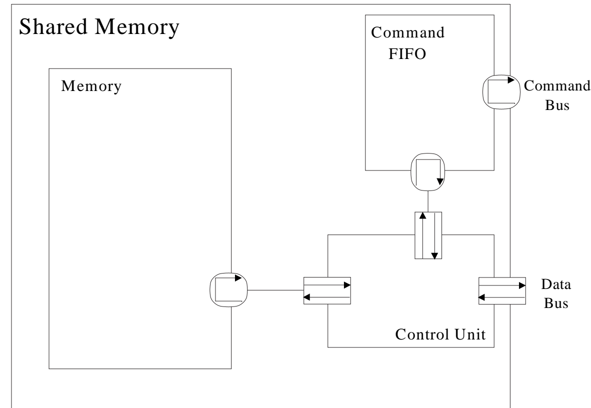

Figure 6.7: Internal Architecture of a Shared Memory

6.4.3.1 Command Bus

When an embedded processor of the platform attempts to perform a request to a shared functional unit, it first has to send a command request to the selected target unit through the command bus. Therefore, the embedded processors act as masters of the command bus and the hardware blocks and shared memories act as slaves, as can be seen in Figure 6.8.

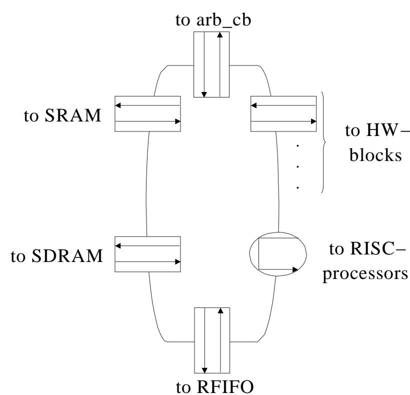

Figure 6.8: Command Bus

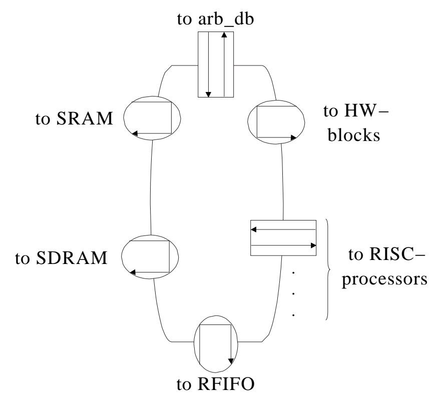

Figure 6.9: Data Bus

6.4.3.2 Data Bus

The command requests are then stored in a receive FIFO inside the functional units and served in the order of arrival. The processing of a command by a functional unit implies a data transmission to/from the read/write transfer registers of the initiator processor. Such data transmissions are then performed through the data bus, which is shared by every functional unit. Contrary to the command bus, the hardware blocks and shared memories act as masters of the data bus and the embedded processors with their respective transfer registers as slaves. A scheme of the data bus is depicted in Figure 6.9.

Chapter 7

Results and Evaluation

7.1 Chapter Introduction

In this Chapter, the procedure followed by the proposed performance estimation methodology is illustrated by means of a case study. The construction of the three models on which the method is based, the Functional, Architectural and Communication model, is explained step by step. This procedure is performed for several architecture–partition alternatives. Subsequently, the results of the method concerning modelling effort, simulation runtime and accuracy are extracted and analysed. At the end of the Chapter, the method results are compared with the results obtained by other system performance estimation techniques.

7.2 Case Study

As already mentioned in Chapter 2.4, the main application area of the present thesis are VLSI networking architectures. Therefore, a packet processor inside a router in a TCP/IP network is taken as a case study for illustrating the complete methodology. Different architecture–partition alternatives are selected and the achieved performance values are extracted and, later on, analysed. Moreover, in order to demonstrate the gains concerning compilation and simulation runtime and to stress the accuracy achieved by the proposed methodology, these metrics are compared with the results obtained by simulating the cycle-accurate co-simulation platform presented in Chapter 6. For this reason, the platform was conceived to cope with TCP/IP packet processing functionalities.

7.2.1 Packet Processor Functionalities

Hosts in a TCP/IP network are bound through routers. The routers determine the path that the packets should follow in order to reach the destination address.

In general, a packet processor inside a linecard receives the data coming from the data link layer and passes it to the network layer. Later on, the packet header corresponding to the network layer is processed and the packet is sent to the next hop, i.e., the next router in the path to the destination address. Packets reach the packet processor from an external interface, the MAC (Medium Access Control) framer. They are processed by the packet processor and stored again in the framer of the corresponding output interface to be sent to the next router.

The main jobs of a packet processor are the following ([86]):

- Header parsing: Fields within the packet header have to be extracted because they contain the information required to decide how the packet will be processed. The parsing needs not be limited to the network layer header, but may also comprise headers of higher OSI layers;
- Classification and Forwarding: A packet has to pass the following processing stages to be classified and routed:
  - Filtering: A certain number of rules assure that only authorised packets pass through the following processing elements;
  - Classification: Context information is assigned to a packet depending on the header fields and according to a set of rules. Accounting and billing facilities as well as QoS-aware packet handling are thus enabled;
  - Next Hop Lookup: Using the destination address field within the packet header, the outgoing link to the next hop is determined;
  - Quality of Service (QoS) Differentiation: Traffic classes are processed with different priorities;
  - Accounting and billing: Traffic statistics are gathered for network engineering, for checking service level agreements and reservations as well as for billing customers according to the current network load and their actual traffic profile.
- Packet Modification: Certain fields within the packet header are modified (e.g., the TTL (Time To Live) is decremented, the new TOS (Type of Service) byte extracted from the classification is inserted and the new CHK (header checksum) is calculated).

- Policing: Once a packet has been classified, its context information is available and, with it, the packet flow to which the packet belongs. Service guarantees can now be checked by verifying Service Level Agreements (SLAs) between customers and the provider of a service for that flow;
- Queueing: After a packet has been admitted for a possible transmission, it must be buffered in the system until it will be either chosen by the link scheduler for transmission or be discarded in case of a congested link;
- Link Scheduling: It is a kind of arbiter that must decide which of the buffered packets will be transferred next through the outgoing link of a networking node.

7.2.2 TCP/IP Packet Processing Data Flow

A subset of the TCP/IP packet functionalities inside the receive functions of a packet processor is taken as a case study. The corresponding data flow is depicted in Figure 7.1. The functions to be implemented are restricted to the user plane functions and comprise *TCP/IP processing v.4 + DiffServ* functionalities such as: packet recognition, packet verification, checksum computation, mark and modification, packet classification, policing and the implementation of the longest prefix match algorithm.

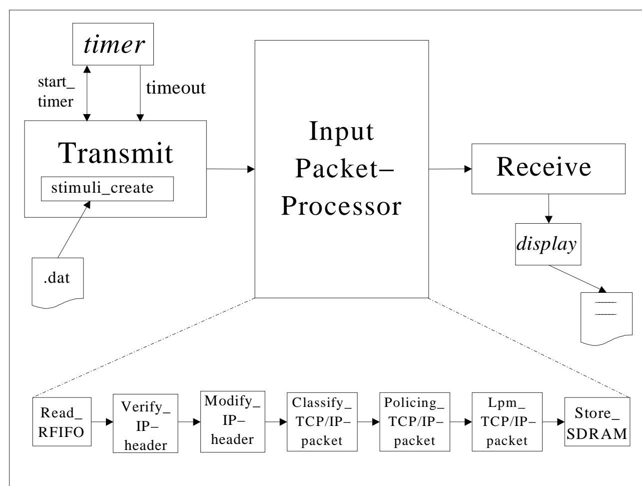

Figure 7.1: Case Study: Input Packet Processing

The packet is first picked up from the receive FIFO (RFIFO) within the interface unit (`read_rfifo`). Further on, the header length, the protocol version and the checksum are

verified (`verify_packet`). Subsequently, the packet is classified (`classify_packet`). Later, the modification of some fields takes place (`modify_packet`), as for example the decrement of the TTL (Time To Live) field and the subsequent new CHK (Checksum) calculation. The policing algorithm is then performed (`policing_packet`) and after obtaining the next hop IP address (performing the LPM (Longest Prefix Match) algorithm) (`lpm_packet`), the packet is finally stored in the packet memory (`store_sdram`).

7.3 Modelling of Architecture–Partition Alternatives

This Section addresses the construction of the three models, Functional, Architectural and Communication ASCSG, for the case study. The goal is to explore different architecture–partition alternatives. For this purpose the Functional model is built up at the beginning and remains unchanged when adding the different architecture–partition configuration information of the alternatives to be tested. The construction of the Architectural and Communication models is automated by means of a configuration file that is fed with the information concerning the different alternatives to be explored.

The single functions are modelled as SystemC modules (the so-called *computation nodes*) with their respective input and output ports and are stored in the library of functions. The initial graph is then built up by collecting the selected *computation nodes* from the library and linking them in the order defined by the specification. The Functional model can comprise multiple instances of the initial graph.

The Functional ASCSG taken as a basis for the exploration is depicted in Figure 7.2.It models the parallel processing of two TCP/IP packets within a packet processor. As already mentioned above, this Functional model then remains unchanged while different architecture–partition alternatives are tested.

As examples of possible architecture–partition solutions, three different alternatives are presented next:

- First Alternative: Towards the creation of the Architectural model for a first alternative, every *computation node* of each packet processing flow, except the *classify_packet* function, are mapped to two different processing units, the embedded processors PE_0 and PE_1. The *classify_packet* function of both packet processing flows are mapped to another processing unit, the hardware block AC_0, in order to accelerate this costly function. This node thus becomes a shared *computation node*. Finally, the *rfifo* and *sdram* functions of both packet processing flows are

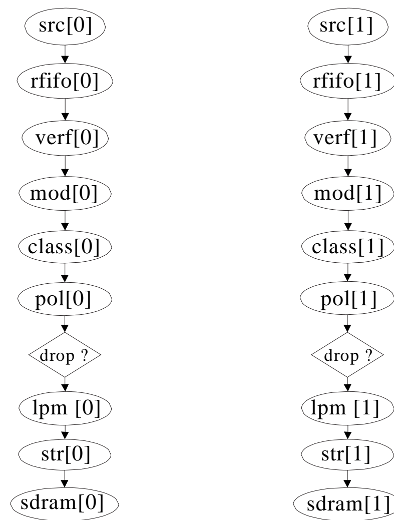

Figure 7.2: Case Study: Functional ASCSG

mapped to the shared interface unit and memory, respectively. These nodes thus also become shared *computation nodes*.

With this mapping information, a generator creates the corresponding configuration file for the Architectural model. For its generation, the corresponding execution time values are extracted from the Mapping Execution Times library (Table 5.1) and are further annotated inside the *computation nodes*. These execution time values have been previously calculated using for it block level performance estimation techniques.

Moreover, in order to avoid access conflicts when attempting to reach a shared *computation node*, an arbitration mechanism, referred to as a *shared element* mechanism ($Shel_j$) as defined in Chapter 4, is automatically added. It is accessed by the corresponding *access nodes* (Acc_{i-j}), which is interleaved between the *computation node* which initiates the read/write request and the shared *computation node*. This mechanism handles each request depending on the priority of the processing unit to which the initiator *computation node* is mapped.

The definition of the communication architecture for the current case study determines a command and a data bus as the communication media. Both the command and data bus are bound to the two embedded processors, to the hardware block and to the interface unit and memory unit. The bandwidth and frequency of both buses are provided as parameters to the communication media.

Each time a transfer between two *computation nodes* mapped to different process-

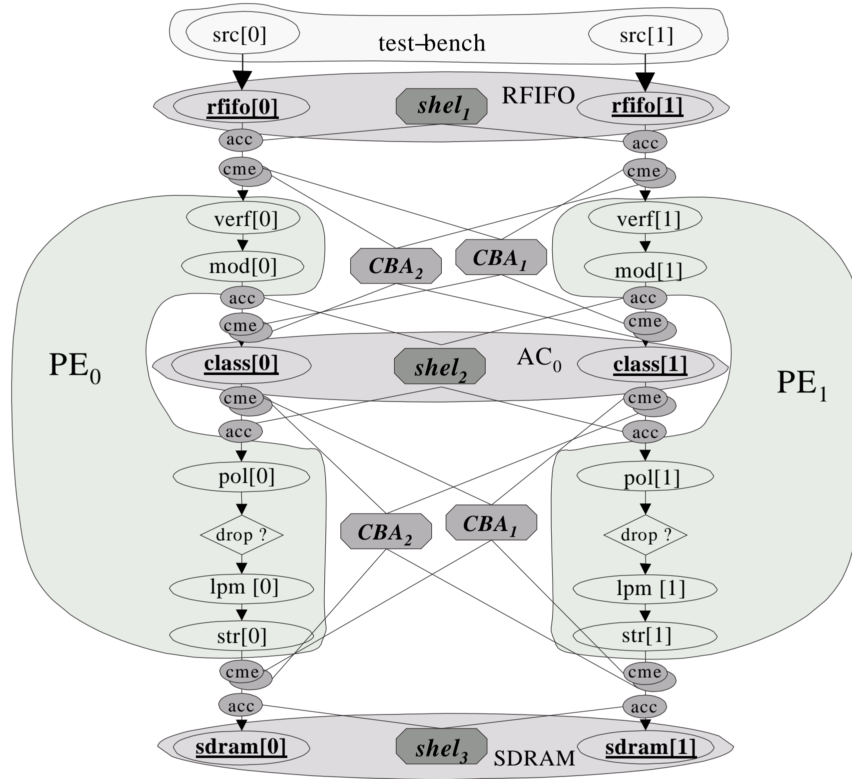

Figure 7.3: First Alternative: Two PEs and One AC

ing units takes place, two sequential *external communication nodes* ($\mathrm{Cme}_{\text{i-j-}}\mathrm{m}_y$) have to be inserted, one mapped to the command bus and the other to the data bus. In the first case, the embedded processors, PE_0 and PE_1, act as masters sending a command request to the selected shared processing unit through the command bus. In the second case, the shared processing units, AC_0, RFIFO and SDRAM, act as masters sending a read/write request to the initiator embedded processor through the data bus. For performing such tasks, both the embedded processors and the shared processing units access the corresponding Command Bus Arbiter (CBA_y). Each CBA_y takes into account the priority of the initiator of the transfer. With this information concerning the communication architecture, its parameters and the mapping of the *communication nodes* to the corresponding communication medium, a generator creates the corresponding configuration file for the Communication model.

The resulting Communication ASCSG for this architecture–partition alternative with two embedded processors and one hardware block accelerating the classification (*classify_packet*) function is depicted in Figure 7.3. For clarity, Figure

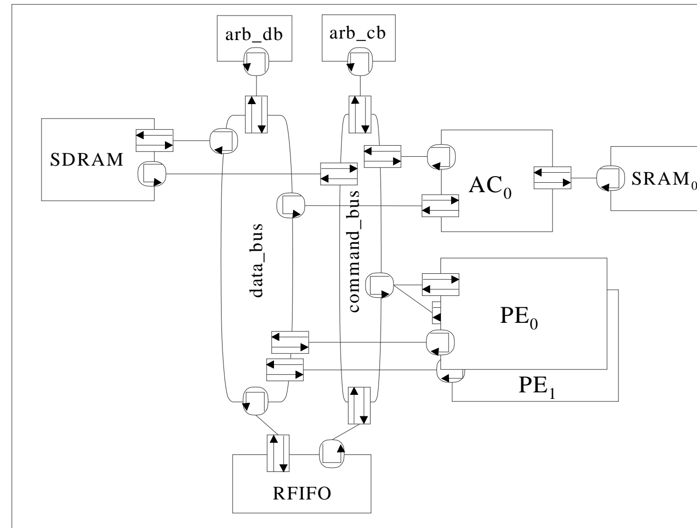

Figure 7.4: Target Architecture First Alternative

7.4 illustrates the structure of the target architecture selected for this alternative. Last Figure also depicts the selected configuration of the cycle-accurate co-simulation platform, whose simulation results will be compared to the results delivered by the ASCSG.

- Second Alternative: By providing a new configuration information for the building of the Architectural and Communication ASCSG, a different architecture–partition alternative for the Functional model described in Figure 7.2 can easily be explored. In this case (Figure 7.5) the longest prefix match algorithm (*lpm_packet* function) for the calculation of the IP address of the next hop is mapped to a second hardware block (AC_1) instead of implementing it inside the embedded processors. This single function is thus sped up. However, a communication overhead is now added. Also for purposes of clarity, Figure 7.6 illustrates the structure of the target architecture selected for this second alternative. Last Figure also depicts the selected configuration of the cycle-accurate co-simulation platform, whose simulation results will be compared to the results delivered by the ASCSG.

- Third Alternative: Again, by providing new configuration information, a different architecture–partition alternative can easily be explored. This third case makes use of the multi-threading capability of one embedded RISC processors (PE_n). Two contexts within an embedded processor carry out the selected packet processing functionalities in parallel, except for the *classify_packet* function that is

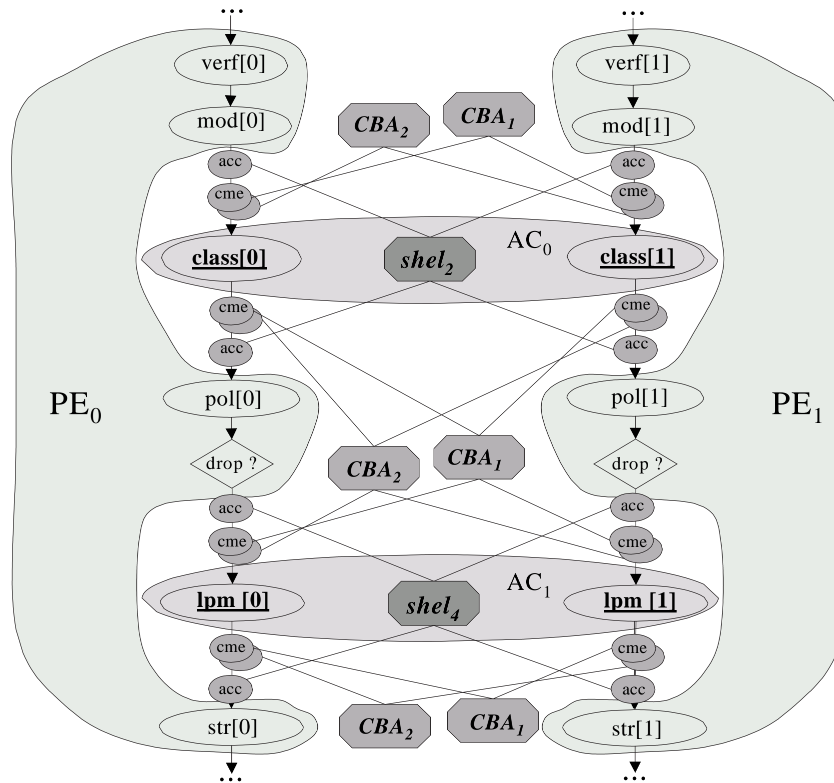

Figure 7.5: Second Alternative: Two PEs and Two ACs

executed externally by a hardware accelerator. Each time a context performs a read/write function from/to an external unit (e.g. RFIFO, Accelerator or SDRAM), the thread is put to sleep and allows the other thread to run if it is ready. Such a mechanism is controlled by the Context Event Arbiter ($\mathrm{CEA_n}$) of the corresponding embedded processor. Once the corresponding external unit notifies the related thread that the request is ready, the external unit signalises (i.e., sends an *event_signal* to the Context Event Arbiter) that the context is ready to run. Nevertheless, it still has to wait until the current active context is put to sleep. As can be seen in Figure 7.7, each time a *context_change* or an *event_signal* takes place, an *internal communication nodes* ($\mathrm{Cmi_{i\text{-}j}\text{-}n_t}$) has to be inserted, which accesses the CEA of the embedded processor to which is mapped.

Combining the mapping of *computation nodes* to the different processing units of the selected target architecture and, in the case of mapping to the embedded processors (PE), to the threads (μthr) supported by each processor, many different architecture–partition alternatives can be tested easily and rapidly. Moreover, the initial Functional model can be enlarged to describe the parallel processing of more packets. The com-

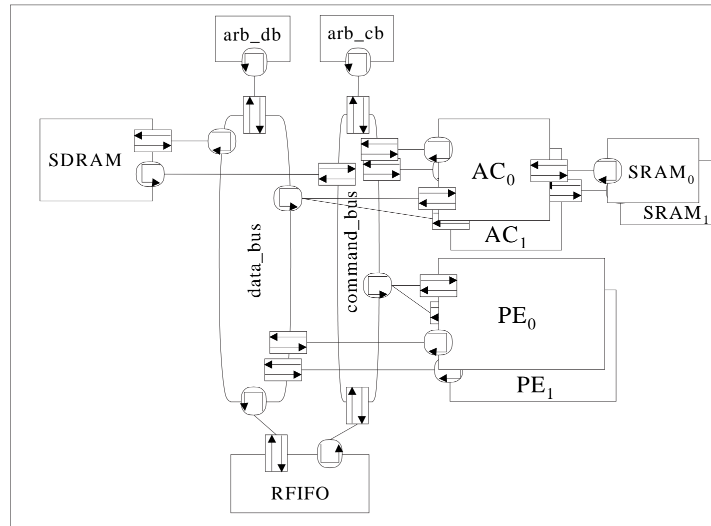

Figure 7.6: Target Architecture Second Alternative

binations show in Table 7.1 have been explored in accordance with the procedure presented above. The number of packets processed in parallel by each test is shown in the first row. In order to get an overview of the order of magnitude, the successive rows provide the number of *access nodes/shared element* mechanisms (acc/shel), *external communication nodes/Command Bus Arbiters* (cme/cba) and *internal communication nodes/Context Event Arbiters* (cmi/cea) to be inserted by the method based on the ASCSG depending on the alternative to be tested.

7.4 Simulation Results

The simulation of the Communication ASCSG for the case study delivers the performance values required for the evaluation of a concrete architecture–partition alternative. For understanding the results obtained, the simulation environment and the predefinitions are introduced first.

7.4.1 Simulation Environment

For the functional validation of the system specification and the further evaluation of the final Communication model, a test-bench is built around the graph. It is responsible for feeding the corresponding model with stimuli and checking the results at the output.

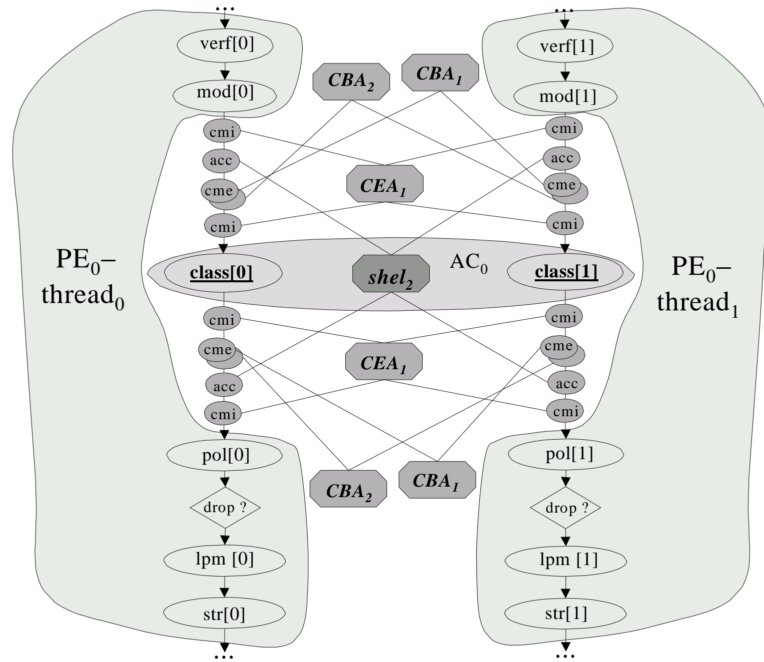

Figure 7.7: Third Alternative: One Multi-Threading PE and One AC

As can be seen in Figure 7.1, the test-bench comprises a module to generate the TCP/IP packets (Transmit) and two modules to receive and visualise the outcoming packets (Receive and Display). The generation module reads the corresponding field values of each packet from a text file to which new packets can easily be added. Then a timer is responsible for sending the generated packets to the output port of the module, and therefore to the input packet processor, with a determined frequency. The interarrival time between consecutive packets is parameterisable, which determines the input throughput. Considering, for instance, a hypothetical 24-byte TCP/IP packet (only the IP header and the two fields, SP (Source Port) and DP (Destination Port), of the TCP header) and an interleaved time of 25 ns, the input throughput is then 915 Mbyte/s. At the output side, the Display module monitors the packets as they reach the last function of the TCP/IP packet processing data flow. The arrival time for each packet is recorded in order to be able to calculate the output throughput.

The same environment, as well as the stimuli and input throughput, are used for the simulation of the cycle-accurate co-simulation platform. In this way, the values obtained from both methods can be compared fairly afterwards.

Table 7.1: Architecture–Partition Alternatives Explored

		1 PE			2 PE		4 PE	
		1 μthr	2 μthr	4 μthr	2 μthr	4 μthr	2 μthr	4 μthr
num_pack		1	2	4	4	8	8	16
no AC	acc/shel	2/2	4/2	8/2	8/2	16/2	16/2	32/2
	cme/cba	4/2	8/2	16/2	16/2	32/2	32/2	64/2
	cmi/cea	2/1	4/1	8/1	8/2	16/2	16/4	32/4
AC_*class*	acc/shel	4/3	8/3	16/3	16/3	32/3	32/3	64/3
	cme/cba	8/2	16/2	32/2	32/2	64/2	64/2	128/2
	cmi/cea	4/1	8/1	16/1	16/2	32/2	32/4	64/4
AC_*class* & AC_*lpm*	acc/shel	6/4	12/4	24/4	24/4	48/4	48/4	96/4
	cme/cba	12/2	24/2	48/2	48/2	96/2	96/2	192/2
	cmi/cea	6/1	12/1	24/1	24/2	48/2	48/4	96/4

7.4.2 Simulation Predefinitions

Before running the simulations, some parameters concerning the different components of the target architecture as well as the values related to the block level performance estimation have to be determined. As an example, the values shown in the following tables are selected for the present case study.

Tables 7.2 and 7.3 show the values for the parameters concerning frequency of the processing units and the frequency, bandwidth and maximum number of longwords (32-bits) to be transmitted at once through each communication medium, the data bus and the command bus.

Table 7.2: Parameter Processing Units

System component	Frequency
RISC processors [PE_n]	200 MHz
HW units [AC_m]	200 MHz

Table 7.4 summarises the performance values of each system function mapped to the processing units of the selected target architecture. The entries marked with an “X” mean that such mapping is not feasible.

Table 7.3: Parameter Communication Media

System component	Frequency	Bandwidth	Max. num. longwords
Data bus	200 MHz	32 bits	6
Command bus	200 MHz	32 bits	1

Table 7.4: Values Block Level Performance Estimation

System function	RISC processor	HW unit
read_rfifo	25 ns	X
verify_packet	1075 ns	X
modify_packet	750 ns	X
classify_packet	1150 ns	125 ns
policing_packet	1275 ns	80 ns
lpm_packet	1750 ns	100 ns
store_sdram	25 ns	X

7.4.3 Output Throughput

In the current case study, which comprises a subset of the TCP/IP packet functionalities inside the receive functions of a packet processor, the processing of each packet ends with its storage in the packet memory (SDRAM). Therefore, for measuring the output throughput achieved by a certain alternative, the arrival time of the incoming packets in the SDRAM are recorded. Then the throughput is calculated at the end of the selected simulation time.

Table 7.5 shows the output throughput for the different architecture–partition alternatives which have been tested. The analysis of the results and the subsequent decisions for modifications are then a task of the designer, who uses the methodology to evaluate different possible implementations.

7.4.4 Performance Values

As already mentioned in previous Chapters, the simulation of the Communication ASCSG delivers the values related to the dynamic behaviour of the system. The evaluation of such values can be very helpful for the designer to find, for instance, where the bottlenecks of a certain alternative are found.

Table 7.5: Output Throughput (packets/s)

		1 PE			2 PE		4 PE	
		1 μthr	2 μthr	4 μthr	2 μthr	4 μthr	2 μthr	4 $\mu\mu$thr
AC	none	147625	151859	150633	302022	299482	603669	598519
	class	154704	168110	165141	333516	330330	664999	662367
	lpm	184191	211903	203740	423355	409741	848812	823715

The proposed methodology is prepared to extract the following performance values from the simulation of the Communication ASCSG: processing time; delay in accessing each shared communication medium; arbitration delay; delay in accessing each shared processing unit; load of each shared processing unit; delay in accessing the process unit by a thread; and the context switch delay. For the extraction of further performance values, the methodology can be easily adapted.

In order to depict these performance values, an architecture–partition alternative is selected from the ones presented in Table 7.1. As an example, the alternative composed of four embedded processors, with hardware support of four threads each, and two hardware accelerators for speeding up the classification and the calculation of the next hop functions is chosen.

The proposed methodology extracts the above-mentioned performance values and delivers them to the designer. Therefore, the values for the selected alternative are then defined and illustrated as an output of the methodology, but are not analysed.

First of all, the queue of requests for both the command and data bus along with the time are depicted in Figures 7.8 and 7.9. The Command Bus Arbiter of the command and data bus stores the requests and serves them when the addressed communication medium becomes free.

The requests waiting to access the accelerators, which are shared by all embedded processors, deliver the trace represented in Figures 7.10 and 7.11. Such requests are stored and handled by the mechanism implemented in the corresponding *shared element* mechanism.

Within a multi-threading embedded processor, the number of contexts which are ready to run at a certain point in time, but that they have to wait until the current active context is put to sleep, can be registered. Until they are served, these requests are buffered in the Context Event Arbiter of the processor, which is responsible for deciding the next context to run. Figures 7.12 and 7.13 show these values for the first and second embedded processor.

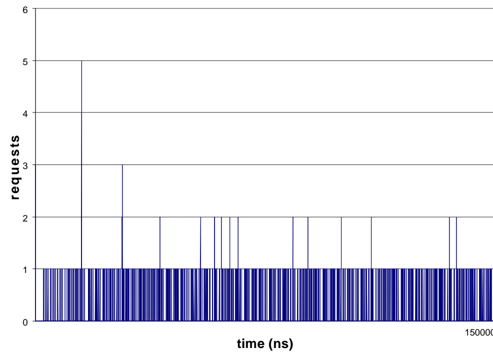

Figure 7.8: Queue Command Bus

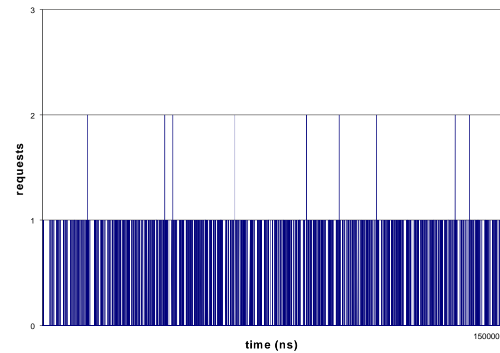

Figure 7.9: Queue Data Bus

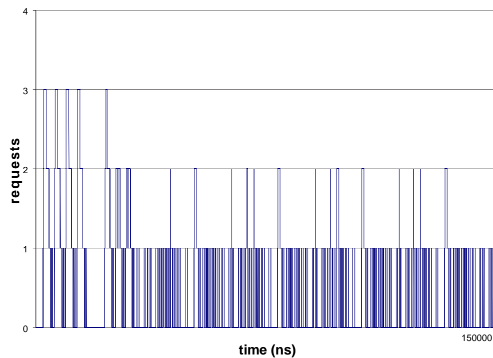

Figure 7.10: Queue Classify Accelerator

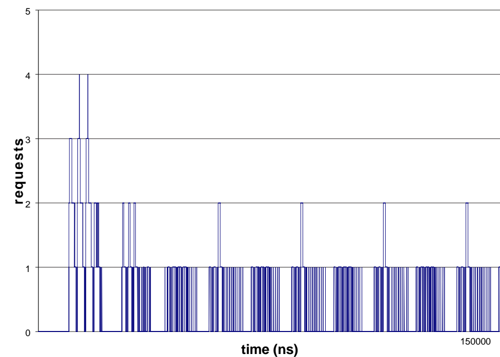

Figure 7.11: Queue LPM Accelerator

The read/write requests to/from a certain communication medium cannot always be handled immediately because the target communication medium is busy at that moment. Moreover, the access to a communication medium is managed by its corresponding Command Bus Arbiter, which takes into account the priority of the initiator of the request. In the present example, the priorities decrease with the increase of the embedded processor and thread number, respectively (i.e., the first thread of the first processor owns the highest priority and the fourth thread of the fourth processor the lowest). The delays when attempting to access the command bus for writting, along with the simulation time, behave as shown in Figure 7.14 for the first thread of each embedded processor and in Figure 7.15 for each thread of the first embedded processor. The delays when attempting to access the data bus for writting are depicted in Figures 7.16 and 7.17.

In the same way, the read/write requests to/from the shared units (accelerators and

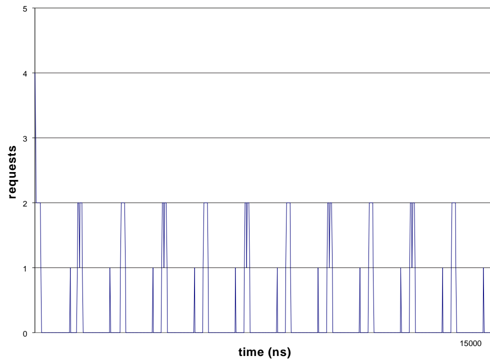

Figure 7.12: Queue Events PE_0

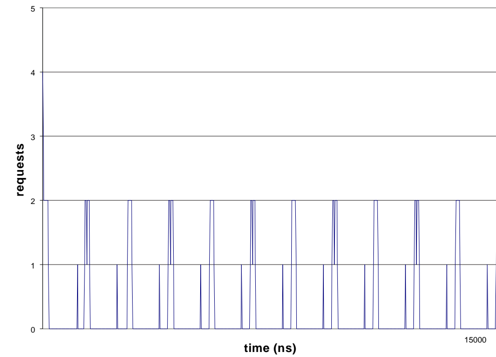

Figure 7.13: Queue Events PE_1

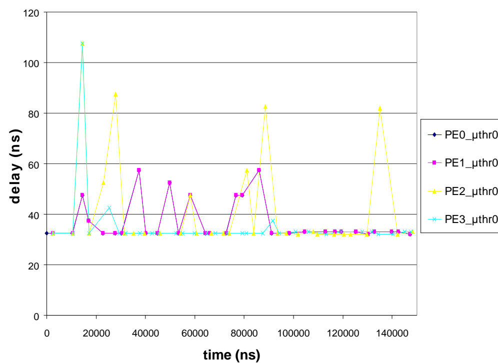

Figure 7.14: Delay Write Request Command Bus; PEs

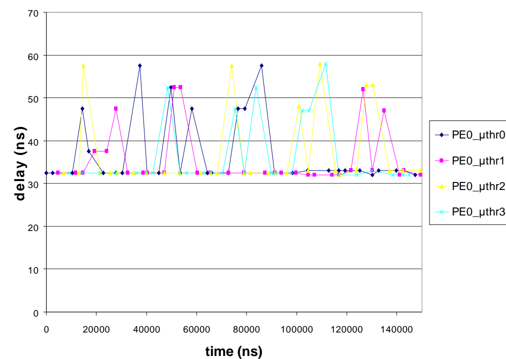

Figure 7.15: Delay Write Request Command Bus; Threads

memory units) exhibit a delay when trying to access the target unit while it is busy. The variations on the write delay depending on the point in time the request arrives is depicted in Figures 7.18 and 7.19 for the shared hardware unit accelerating the classification function, and in Figures 7.20 and 7.21 for the selected hardware unit to speed-up the calculation of the next hop.

Within a multi-threading embedded processor, only one thread of execution gets the control of the processing unit at a time. This means that the other threads have to wait until the current thread is put to sleep. This wait time is depicted in Figures 7.22 and 7.23 for every thread within the first and second processors, respectively.

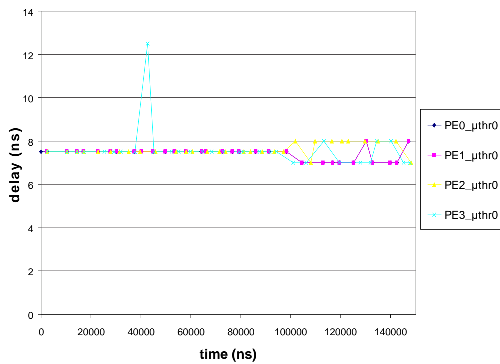

Figure 7.16: Delay Write Request Data Bus; PEs

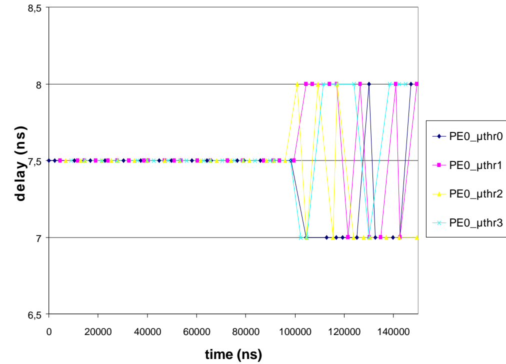

Figure 7.17: Delay Write Request Data Bus; Threads

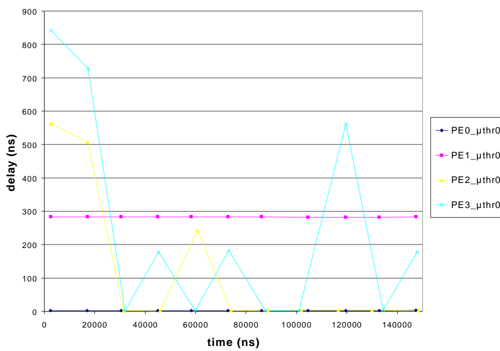

Figure 7.18: Delay Write Request Classify; PEs

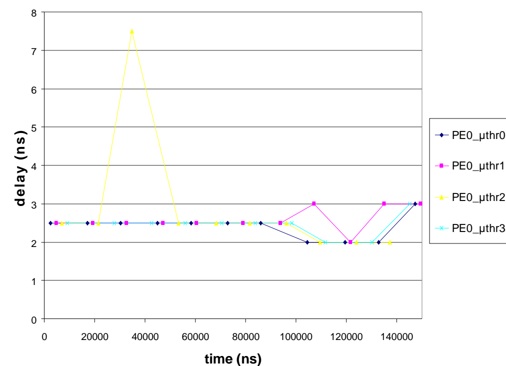

Figure 7.19: Delay Write Request Classify; Threads

7.5 Simulation Speed of the ASCSG

Once the Communication ASCSG for the different architecture–partition alternatives explored has been built, the corresponding SystemC file (`main_communication.cpp`) for each alternative is compiled and linked with the SystemC core and specific libraries, and the related executable file is created. The execution of such a file corresponds to the simulation of the corresponding model.

The simulations have been carried out on a notebook with a mobile AMD Athlon XP 2000+ processor (frequency 1667 MHz) under the Linux operating system.

The second column of Table 7.6 shows the values concerning the elaboration plus simulation time for each architecture–partition alternative explored by simulating the

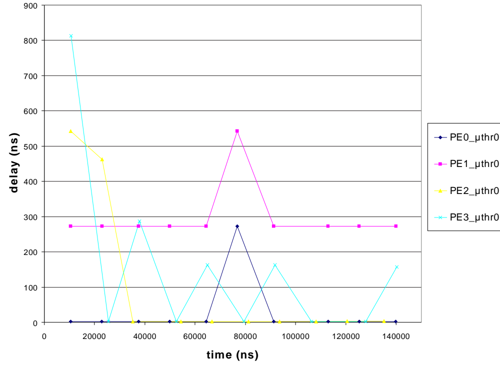

Figure 7.20: Delay Write Request LPM; PEs

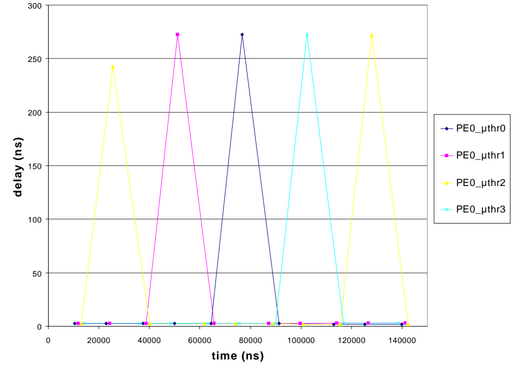

Figure 7.21: Delay Write Request LPM; Threads

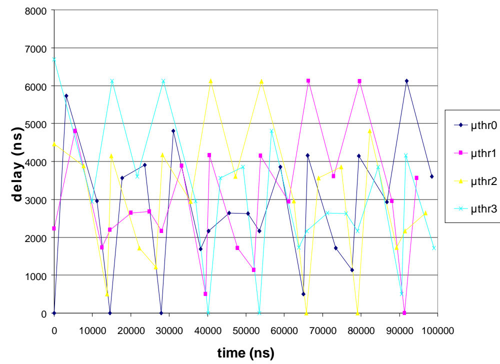

Figure 7.22: Delay Wait Event PE_0

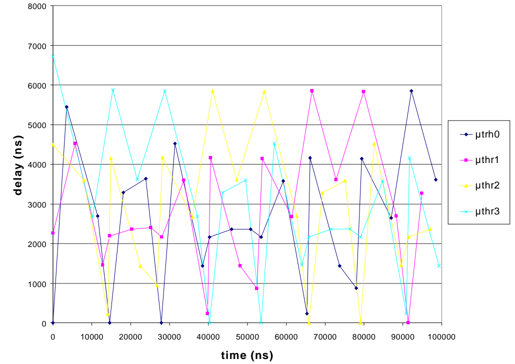

Figure 7.23: Delay Wait Event PE_1

corresponding Communication ASCSG. Furthermore, the same architecture–partition alternatives have been tested using the cycle-accurate co-simulation platform presented in Chapter 6. The elaboration plus simulation time achieved by the cycle-accurate simulations are presented in the third column. Both models have been stimulated with the same input throughput of 915 Mbyte/s during 200,000 ns.

Lastly, the fourth column shows the decrease of simulation time achieved by the proposed method based on the ASCSG, taking as a basis the results of the cycle-accurate model. The equation used for the calculation of the gain in terms of simulation time is given by Formula 7.1.

$$Gain = \frac{T_{\text{sim}}[Cycleacc.] - T_{\text{sim}}[ASCSG]}{T_{\text{sim}}[Cycleacc.]} * 100 \qquad (7.1)$$

Table 7.6: Simulation Speed

			ASCSG (sec)	Cycle acc.(sec)	Gain (%)
1 PE	2 μthr				
		0 AC	1.6	5.12	68.75
		1 AC	0.93	2.42	61.57
		2 AC	5.62	15.71	64.23
	4 μthr				
		0 AC	3.02	9.36	67.74
		1 AC	1.73	5.03	65.61
		2 AC	8.92	28.8	69.03
2 PE	2 μthr				
		0 AC	2.93	8.34	64.87
		1 AC	1.76	7.91	77.75
		2 AC	8.35	27.05	69.13
	4 μthr				
		0 AC	6.03	17.92	66.35
		1 AC	3.66	12.87	71.56
		2 AC	25.19	68.96	63.47
4 PE	2 μthr				
		0 AC	6.05	21.71	72.13
		1 AC	3.7	10.65	65.26
		2 AC	24.03	77.39	68.95
	4 μthr				
		0 AC	14.13	40.09	64.75
		1 AC	14.01	43.63	67.89
		2 AC	69.62	239.63	70.95

The results demonstrate an increase of the simulation speed around 70% when applying the method based on the ASCSG as compared to the cycle-accurate model. Moreover, the compilation time of the cycle-accurate model exceeds the compilation time of the ASCSG model by an order of magnitude of 20.

These results confirm the expectations concerning the acceleration of the simulation when increasing the level of abstraction of the modelling. For complex systems, this reduction of compilation and simulation time will allow the exploration of more architecture–partition alternatives than when a model is applied at a lower level of abstraction. Nevertheless, for a fast exploration procedure, is the modelling effort required prior the simulation a more decisive factor (see Section 7.7).

7.6 Accuracy of the ASCSG

In order to determine the accuracy of the results delivered by the proposed methodology, the output throughput is taken as a performance value for performing the comparison with the cycle-accurate model. The simulation of the Communication ASCSG is compared with the output throughput achieved by the cycle-accurate co-simulation platform for the different architecture–partition alternatives explored. Both models have been stimulated with the same input throughput of 915 Mbyte/s during 200,000 ns.

Table 7.7 shows the output throughput achieved by both models (the second column displays the results achieved by the ASCSG and the third shows the results delivered by the cycle-accurate model) together with the deviation (fourth column) introduced by the ASCSG, using the cycle-accurate results as a basis for the comparison. The equation used for the calculation of the deviation is given by Formula 7.2.

$$Deviation = \frac{Output_throughput[Cycleacc.] - Output_throughput[ASCSG]}{Output_throughput[Cycleacc.]} * 100 \tag{7.2}$$

The observation of the fourth column of the table indicates a low deviation of around 1.5% overestimation for the performance results delivered by the ASCSG. Thus, it can be concluded that the method based on the ASCSG delivers highly accurate results. This makes the proposed methodology reliable to be used for fair performance estimations early in the design. This overestimation delivered by the ASCSG can be explained considering the non-deterministic behaviour of the ISS (Instruction Set Simulator of the cycle-accurate platform) with respect to external events due to its five-stage pipeline. On the contrary, the ASCSG is modelled to react immediately to external events.

This achievement has been possible by considering the internal communication of the system and by providing a mechanism for solving conflicts when accessing the shared communication media. Moreover, the consideration of the hardware multi-threading support by the embedded processors has led to accurate results by providing an arbiter mechanism that controls the context switches within such processors.

7.7 Modelling Effort of the ASCSG

When studying and comparing the speed by analysing different architecture–partition alternatives, more must be taken into consideration than the simulation time alone.

Table 7.7: Simulation Accuracy

			ASCSG (pck/sec)	Cycle acc.(pck/sec)	Deviation (%)
1 PE	2 μthr				
		0 AC	152,880	151,859	-0.67
		1 AC	168,528	168,110	-0.25
		2 AC	212,976	211,903	-0.51
	4 μthr				
		0 AC	152,659	150,633	-1.34
		1 AC	166,649	165,141	-0.91
		2 AC	209,147	203,740	-2.65
2 PE	2 μthr				
		0 AC	304,080	302,022	-0.68
		1 AC	335,922	333,516	-0.72
		2 AC	425,547	423,355	-0.52
	4 μthr				
		0 AC	301,753	299,482	-0.76
		1 AC	332,504	330,330	-0.66
		2 AC	419,811	409,741	-2.46
4 PE	2 μthr				
		0 AC	607,913	603,669	-0.70
		1 AC	668,384	664,999	-0.51
		2 AC	853,415	848,812	-0.54
	4 μthr				
		0 AC	603,068	598,519	-0.76
		1 AC	665,573	662,367	-0.48
		2 AC	836,121	823,715	-1.51

More importantly, the modelling effort required prior to the simulation also has to be taken into account. When applying a structural model of the system under study for design space exploration, the analysis of different architecture–partition alternatives demands, in most cases but not always, a re-building of the structure of the model. The modelling effort associated to such re-building depends on the type of action required. If only an addition of an existing block is performed, the effort is generally not very high. However, if a new block has to be modelled and linked to the system architecture, the modelling and adjustment can demand considerable effort. Therefore, it can be concluded that the effort and, consequently, time needed for such re-building makes the exploration of many different alternatives nearly infeasible in many cases.

The proposed methodology offers a solution to this problem by applying a model based on a process graph that remains unchanged during the exploration of different architecture–partition alternatives. Once the functional specification, the architecture structure and resources, and the communication media are defined, the proposed methodology offers a low modelling effort when testing different partition alternatives. The configuration information is then provided for each alternative to be tested. Figure 7.24 depicts the few necessary parameters that are comprised in the configuration information for each alternative. First of all, the Architectural model requires the new mapping of functions onto the available processing units and the scheduling information. Secondly, the Communication model needs the new mapping of the external communication transfers onto the selected communication media and of the internal communication transfers to the corresponding multi-threading embedded processors. Following this procedure, many alternatives can be explored by providing only the corresponding configuration information.

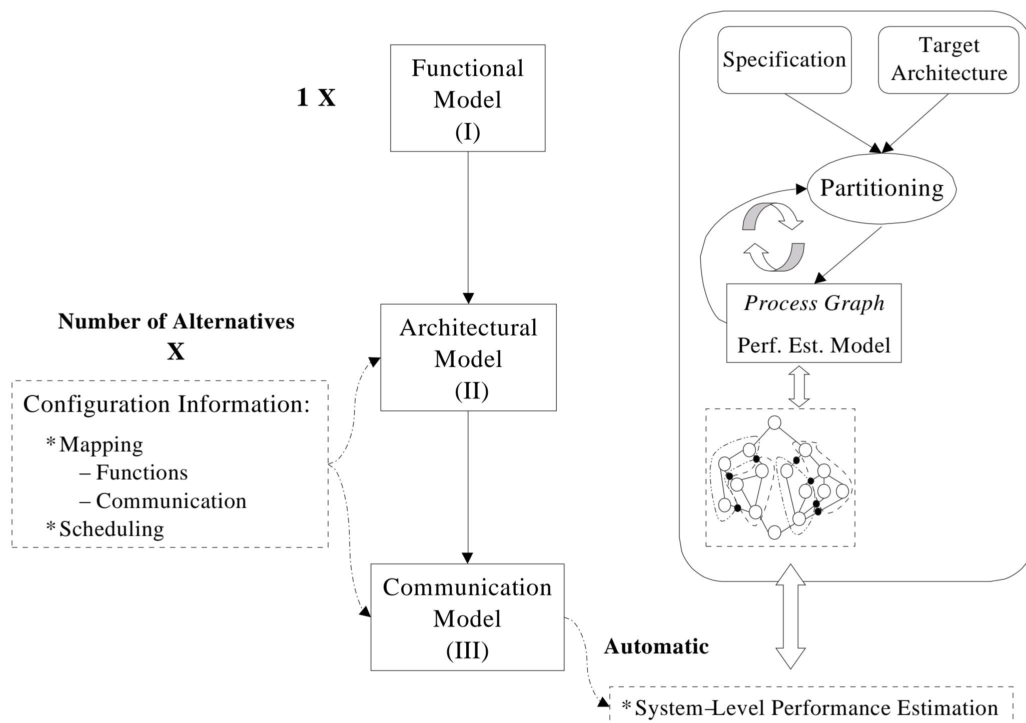

Figure 7.24: Modelling Effort

7.8 Comparison with Other Performance Estimation Methods

Different approaches towards the estimation of the performance achieved by a certain architecture–partition alternative can be found in the literature (see Chapter 2.3). They address the problem at different levels of abstraction and they use different techniques to solve it, generally reaching a compromise among different parameters, as for example the modelling effort, the exploration time and the accuracy of the results.

The methodology proposed in this thesis covers some of the weaknesses of the existing approaches up to date. In particular, the new method based on the Annotated SystemC Conditional Synchronisation Graph (ASCSG) addresses the problem concerning the re-building effort, i.e., the modelling effort necessary each time a new partition alternative is to be tested. This is a decisive factor if many alternatives have to be explored.

Three representative performance estimation methods at system level have been selected and a comparative study, together with the proposed methodology, has been performed. The results can be seen in Table 7.8. These three selected approaches address different techniques to deliver the performance of the system under study.

Next, the main advantages and disadvantages of each method are determined and, later on, they are compared with the enhancements of the new methodology based on the ASCSG.

The performance analysis method proposed by K. Lahiri (ECE Department, University of California, San Diego) in [66] constitutes a hybrid trace-based technique only suitable for the design of on-chip communication architectures. During the partitioning decision, the communication architecture is not taken into account and, moreover, the partitioning of the functionalities remains unchanged when exploring the communication structure. This approach achieves an acceptable accuracy (around 4%). However, a high modelling effort is required for the first phase.

The main goal of the approach suggested by A. Baghdadi (TIMA Laboratories, Grenoble) ([79]) is the achievement of a fast architecture exploration loop. It leads to a trade-off between speed and accuracy, where a variable deviation of around 10% makes the results fairly unreliable.

Third, the approach proposed by L. Thiele (ETH Zurich) in [69] consists of an analytical procedure for the estimation of performance achieved by packet processors. This allows a fast exploration of several alternatives with a very low re-building effort, but the results highly differ from a real implementation. For this reason, further estimations at lower levels of abstraction have to be performed. Moreover, the results

Table 7.8: Comparison Performance Estimation Approaches

	K. Lahiri ([66])	A. Baghdadi ([79])	L. Thiele ([69])	ASCSG
Classification	Trace-based	Sim.-based and static	Analytical	Sim.-based
Scope	Selection communication architecture	Fast architecture exploration loop	Selection architecture of packet processors	Fast reconfigurable system perf. estimation
Level of abstraction	RTL and System	RTL and System	System	System
Language	1^{st}ph.: Esterel 2^{nd}ph.: C/C++	SDL	C/C++	SystemC
HW–SW co-simulation	POLIS & PTOLEMY	MUSIC	—	—
Modelling effort	1^{st}ph.: high 2^{nd}ph.: very low	RTL: high Explor.: low	low	low
Accuracy	around 4%	around 10%	low	around 1.5%
Simulation time	1^{st}ph.: long 2^{nd}ph.: very short	RTL: very long Explor.: short	medium Linear.: short	short

lose accuracy when the necessary curves (arrival curves for describing packet rates and service curves for describing bounds on the computation capability of the resources) have to be linearised for reaching a faster simulation.

The proposed method delivers fast simulations comparable to the speed of the second phase of Lahiri's technique and to the speed of the exploration loop of Baghdadi's approach. At the same time, the accuracy of the ASCSG results is noticeably higher than the ones delivered by Thiele and by Baghdadi. Even more important when the number of alternatives to be tested is high is the modelling effort required prior to the simulation. By applying the ASCSG, the modelling effort is comparable to the low effort required by the procedure from Thiele. Lastly, from the point of view of the applicability, the spectrum covered by the ASCSG is wider than the spectrum of Lahiri (only applicable to select the on-chip communication architecture) and of Thiele (intended for the architecture selection of packet processors). In principle, the ASCSG is applicable to every control-dominated system with the only restriction of not containing infinite loops in the initial functional specification.

In summary, it can be said that the new methodology offers the advantages concerning modelling effort, simulation time and accuracy of the results that are individually provided by the existing techniques. Moreover, it emphasises the problem concerning the rebuilding effort, i.e., the modelling effort necessary each time a new partition alternative is to be tested.

Chapter 8

Summary and Conclusions

The increasing size, speed and complexity of the latest designs makes that the classical hardware–software co-design procedures do no longer meet the design requirements posed by today's complex systems. A higher level of abstraction than RTL is needed. At system level, the designer has more freedom to explore the trade-offs and detect the bottlenecks. This would be very costly at lower levels of abstraction.

The challenges introduced by the design space exploration of complex systems pass through the exploration of a large number of architecture–partition alternatives. This challenge requires both a fast performance estimation methodology with a low modelling effort and an exploration strategy for narrowing down the possible solutions. This will allow the designer to cope with the large number of alternatives and find the one that optimally meets the requirements.

In general, the estimation of the performance achieved by a certain architecture–partition alternative involves two steps: the modelling of the specification and the architecture; and the evaluation of the performance. Concerning the first step, the system functionalities and the components which make up the architecture can be represented by applying some formal representation, such as deterministic graphs and analytical models, or by building up a structural model of the target architecture and mapping the functionalities onto the architecture. This last technique delivers a precise estimation, but, most of the times, the time and effort it takes to try a new alternative is considerable. On the other hand, describing the functionalities in terms of a graph and adding the relevant information of the target architecture to the graph, the simulation time and effort towards a re-partitioning is less costly and consequently more partition alternatives can be simulated and evaluated. After choosing a partitioning that meets the performance constraints, the way towards synthesis (back-end) goes through a structural model of the target architecture. This step is costly in terms of design,

but it now has to be done only once. The proposed methodology addresses this last approach.

In this thesis, a novel methodology for the estimation of the performance of System-on-Chip solutions has been presented. An Annotated SystemC Conditional Synchronisation Graph (ASCSG) is built up for this purpose. This graph comprises the functional specification of the system and is further annotated with information concerning the target architecture and the selected partitioning of the functionalities onto the available resources. Furthermore, the required mechanisms to solve resource contentions are added to the graph. The simulation of this graph covers the dynamic behaviour of the system and delivers its performance estimation. It is particularly significant that the modelling effort when applying the proposed method is considerably lower than when a structural model of the target architecture is built up at a lower level of abstraction. At the same time, the new methodology speeds up the simulation and enhances the accuracy of the results of existing approaches. These characteristics allow a fast and easy exploration of several alternatives.

The implementation of the method is based on the system level language SystemC. It is noteworthy that a novel usage of this language has been applied for the description and further annotation of the ASCSG. Existing approaches have applied SystemC to describe the system functionalities and the target architecture in a structural way. SystemC characteristics such as the modelling of time, reactivity and concurrency, and its help in evaluating resource contentions, make this language especially suitable for the implementation of the methodology.

The proposed method is oriented towards supporting the design of multi-processing, multi-threading architectures for networking applications. The new challenges with regard to speed (hardware implementation) and flexibility (software implementation) pursued by the most recent networking architectures introduces in their design a scenario of multiple alternatives. In order to find the best compromise between hardware and software, the exploration of many architecture–partition alternatives is required. Moreover, from the point of view of the user of such multi-processing, multi-threading architectures, the methodology also plays an important role. In this case, the flexibility in terms of mapping of the functions onto the fixed processing units and threads opens a large scenario of alternatives. The decision concerning the optimal mapping which meets the requirements is facilitated if a model of the system at a high-level of abstraction is provided.

The application of the proposed methodology has been illustrated by means of a case study from the networking world. Different architecture–partition alternatives have been explored and the results have been shown. The main criteria selected for the evaluation of the methodology have been: the modelling effort required prior the

simulation when exploring a new alternative; the duration of the simulation for each alternative; and the accuracy of the performance results obtained. Together, these three criteria are the basis for a fast, reconfigurable and accurate performance estimation methodology.

Concerning the modelling effort, the proposed methodology based on the ASCSG requires a low re-building effort when a new partition alternative is to be tested. It only requires the new configuration information concerning mapping and scheduling for generating the executable model, whose simulation delivers the performance values.

For the calculation of the other two criteria, the executable models for the alternatives tested have been run and the simulation time and output throughput have been extracted. As a basis for the comparison, the results of cycle-accurate platform have been used, where both metrics have also been measured. The analysis of the results demonstrates a considerable reduction of simulation time when applying the method based on the ASCSG (around 70%), whereas the accuracy remains within an acceptable tolerance (around 1.5% overestimation).

Next, the restrictions encountered during the development of the proposed methodology are summarised and some ideas for extending them are given. They can be the subject of future work.

First, the design of VLSI networking architectures has been selected as the main application scenario. Nevertheless, further control-dominated application scenarios with data and control dependencies can be, in principle, explored by applying the proposed method. A restriction to this generalisation is encountered in application scenarios whose specifications contain infinite loops. For such cases, a graph-based representation able to capture both data and control dependencies and mainly intended for handling loops, could be used as a basis for the development of a similar methodology.

The inter-components communication architecture has been restricted to point-to-point and bus-based System-on-Chip architectures. The extension of the methodology to cope with other kinds of on-chip communication structures, such as an on-chip crossbar switch ([107]), can be considered in the future.

A third aspect that has been left open is the extension of the methodology to cover other scheduling algorithms besides the ASAP (As Soon As Possible) algorithm. Either a dynamic or a static scheduling algorithm could be implemented depending on the characteristics of the application scenario. A kind of control unit or an RTOS (Real Time Operating System) constitute two possible realisations for implementing the selected scheduling algorithm.

Note that the integration of the new method in a design environment capable of discriminating among partition alternatives depending on their performance has not been implemented yet. Therefore, a further extension of this thesis would be the integration of the method inside a design space exploration environment, where the new alternatives to be tested are automatically chosen.

After summarising the most important issues covered in this thesis, it can be concluded that the proposed methodology solves the shortcomings concerning modelling effort, simulation time and accuracy of the results, which are individually present in existing approaches. Especially significant is that the new methodology offers a low rebuilding effort before testing new architecture–partition alternatives, a fast simulation runtime and an automation of the procedure. Moreover, the accuracy of the performance results is maintained within an acceptable tolerance thanks to the consideration of the internal communication of the system.

Bibliography

[1] *Semiconductor Industry Association (SIA) home page*, http://www.semichips.org/home.cfm.

[2] *International Technology Roadmap for Semiconductors (ITRS) home page*, http://public.itrs.net.

[3] J.Plantin, E.Stoy, *Aspects on System-Level Design*, Proc. 7th International Workshop on HW/SW Codesign (CODES/CASHE'99), Rome, Italy, May 1999.

[4] B.Batley, R.Klein, S.Leef, *Hardware/Software Co-Simulation Strategies for the Future*, Menthor Graphics Corporation. White Paper

[5] P.Alexander, P.Flake, B.Bailey, C.Eisner, W.Rosenstiel, M.Fujita, *Who Cares About System Verification?*, Panel Session, Forum on Specification and Design Languages (FDL'02), Marseille, France, September 2002.

[6] J.Gerlach, W.Rosenstiel, *System Level Design Using the SystemC Modeling Platform*, Proc. Workshop on System Design Automation (SDA'01), Dresden, Germany, March 2000.

[7] A.Gerstlauer, R.Dömer, J.Peng, D.J.Gajski, *System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers, 2001.

[8] M.Jersak, D.Ziegenbein, F.Wolf, K.Richter, R.Ernst, *Embedded System Design Using the SPI Workbench*, Proc. Forum on Specification and Design Languages (FDL'00), Tübingen, Germany, September 2000.

[9] N.Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, 1993.

[10] *ACCELLERA home page*, http://www.accelera.org.

[11] *Rosetta home page*, http://www.sldl.org.

[12] T.Grötker, S.Liao, G.Martin, S.Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[13] *Electronic Design Automation home page*, http://www.edaltd.co.uk.

[14] *CoCentric SystemC Compiler home page*, http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC_ds.html.

[15] *Forte Design Systems home page*, http://www.forteds.com.

[16] *ODETTE home page*, http://odette.offis.de.

[17] *SystemC-plus home page*, http://odette.offis.de/systemc-plus/systemc-plus.php.

[18] *SpecC*, http://www.ics.uci.edu/ specc/index.html.

[19] Daniel D.Gajski, Jianwen Zhu, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

[20] *SDL home page*, http://www.sdl-forum.org.

[21] *Esterel home page*, http://www-sop.inria.fr/meije/esterel/esterel-eng.html.

[22] *UML home page*, http://www.omg.org/uml.

[23] *SystemVerilog home page*, http://www.eda.org/sv-ec.

[24] G.DeMicheli, M.Sami *Hardware/Software Co-Design*, Kluwer Academic Publishers, 1996.

[25] R.Niemann, *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*, Kluwer Academic Publishers, 1998.

[26] *PTOLEMY home page*, http://ptolemy.eecs.berkeley.edu.

[27] R.Camposano, J.Wilberg, *Embedded System Design*, Design Automation for Embedded Systems, vol.1, no.1-2, pp.5-50, 1996.

[28] *COSYMA home page*, http://www.ida.ing.tu-bs.de/research/projects/home.e.shtml

[29] R.K.Gupta, G.De Micheli, *A Co-Synthesis Approach to Embedded System Design Automation*, Design Automation for Embedded Systems, vol.1, no.1-2, pp.69-120, 1996.

[30] *POLIS home page*, http://www-cad.eecs.berkeley.edu/ polis.

[31] *CHINOOK home page*, http://www.cs.washington.edu/research/chinook.

[32] *Cadence home page*, http://www.cadence.com.

[33] *Cadence Virtual Component Co-Design (VCC) home page*, http://www.cadence.com/products/vcc.html.

[34] *Summit home page*, http://www.sd.com.

[35] P.Bjureus, A.Jantsch, *Performance Analysis with Confidence Intervals for Embedded Software Processes*, Proc. 14th International Symposium on System Synthesis (ISSS'01), Montreal, Quebec, Canada, October 2001.

[36] J.K.Suzuki, A.Sangiovanni-Vincentelli, *Efficient Software Performance Estimation Methods for Hardware/Software Codesign*, Proc. 33rd Design Automation Conference (DAC'96), Las Vegas, USA, June 1996.

[37] J.R.Bammi, E.Harcourt, W.Krujitzer, L.Lavagno, M.T.Lazarescu, *Software Performance Estimation Strategies in a System-Level Design Tool*, Proc. 8th International Workshop on HW/SW Codesign (CODES'00), San Diego, USA, May 2000.

[38] J.Liu, M.Lajolo, A.Sangiovanni-Vincentelli, *Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator*, Proc. 6th International Workshop on HW/SW Codesign (CODES/CASHE'98), Seattle, USA, 1998.

[39] M.Lajolo, M.Lazarescu, A.Sangiovanni-Vincentelli, *A Compilation-Based Software Estimation Scheme for Hardware/Software Co-Simulation*, Proc. 7th International Workshop on HW/SW Codesign (CODES/CASHE'99), Rome, Italy, May 1999.

[40] J.Gong, D.Gajski, S.Narayan, *Software Estimation from Executable Specifications*, Technical Report ICS-93-5. March 1993.

[41] S.Malik, M.Martonosi, *Static Timing Analysis of Embedded Software*, Proc. 34th Design Automation Conference (DAC'97), Anaheim, USA, June 1997.

[42] J.Henkel, R.Ernst, *High-Level Estimation Techniques for Usage in Hardware/Software Co-Design*, Proc. Asia South Pacific - Design Automation Conference (ASP-DAC'98), Yokohama, Japan, February 1998.

[43] S.Narayan, D.D.Gajski, *Area and Performance Estimation from System-Level Specifications*, University of California Irvine, Dept. of Information and Computer Science, Technical Report ICS-92-16. 1992.

[44] F.Vahid, D.Gajski *Incremental Hardware Estimation During Hardware/Software Functional Partitioning*, IEEE Transactions on VLSI Systems, vol.3, no.3, p.459.464, September 1995.

[45] J.Henkel, R.Ernst, *A Path-Based Technique for Estimating Hardware Runtime in HW/SW-CoSynthesis*, Proc. IEEE/ACM 8th International Symposium on System Synthesis (ISSS'95), Cannes, France, September 1995.

[46] J.Henkel, *Automatisierte Hardware/Software Partitionierung im Entwurf integrierter Echtzeitsysteme*, Ph. Dissertation submitted to the Technische Universität Carolo-Wilhelmina zu Braunschweig, Germany, 1996.

[47] P.V.Knudsen, J.Madsen, *Integrating Communication Protocol Selection with Hardware/Software Codesign*, Proc. IEEE/ACM 8th International Symposium on System Synthesis (ISSS'95), Cannes, France, September 1995.

[48] K.Hines, G.Borrielo, *Optimizing Communication in Embedded System Co-Simulation*, Proc. 5th International Workshop on HW/SW Codesign (CODES/CASHE'97), Braunschweig, Germany, 1997.

[49] K.Hines, G.Borrielo, *Dynamic Communication Models in Embedded System Co-Simulation*, Proc. 34th Design Automation Conference (DAC'97), Anaheim, USA, 1997.

[50] J.Davis, C.Hylands, J.Janneck, E.A.Lee, J.Liu, X.Liu, S.Neuendorffer, S.Sachs, M.Stewart, K.Vissers, P.Whitaker, Y.Xiong, *Overview of the Ptolemy Project*, Technical Memorandum UCB/ERL M01/11. March 2001.

[51] B.D.Theelen, J.P.M. Voeten, L.J.van Bokhoven, G.G.de Jong, A.M.M.Niemegeers, P.H.A.van der Putten, M.P.J. Stevens, J.C.M.Baeten, *System-Level Modelling and Performance Analysis*, Proc. STW/IEEE 1st PROGRESS Workshop, Katzow, Germany, September 2000.

[52] J.A. Rowson, A.Sangiovanni-Vincentelli, *Interface-Based Design*, Proc. 34th Design Automation Conference (DAC'97), Anaheim, USA, 1997.

[53] C.P.Joshi, *A New Performance Evaluation Approach for System Level Design Space Exploration*, Proc. 15th International Symposium on System Synthesis (ISSS'02), Kyoto, Japan, 2002.

[54] M.Gasteier, M.Glesner, *Bus-Based Communication Synthesis on System Level*, ACM Transactions on Design Automation of Electronic Systems, pp. 1-11, January 1999.

[55] S.Dey, S.Bommu, *Performance Analysis of a System of Communicating Processes*, Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD 97), San Jose, USA, November 1997.

[56] P.V.Knudsen, J.Madsen, *Graph-Based Communication Analysis for Hardware/Software Codesign*, Proc. 7th International Workshop on HW/SW Codesign (CODES/CASHE'99), Rome, Italy, 1999.

[57] P.Pop, P.Eles, Z.Peng, *Performance Estimation for Embedded Systems with Data and Control Dependencies*, Proc. 8th International Workshop on HW/SW Codesign (CODES'00), San Diego, USA, May 2000.

[58] P.Eles, A.Doboli, P.Pop, Z.Peng, *Scheduling with Bus Access Optimization for Distributed Embedded Systems*, IEEE Transactions on VLSI Systems, vol.8, no.5, pp. 472-491, October 2000.

[59] D.A.Patterson, J.L.Hennessy, *Computer Architecture: A Quantitative Approach*, San Mateo, CA, USA. Morgan Kaufman, 1989.

[60] *WARTS: Wisconsin Architectural Research Tools Set, Comput. Sci. Dept. Univ. Wisconsin*, http://www.cs.wisc.edu/larus/warts.html

[61] P.Liverse, P.v.Wolf, E.Deprettere, *A Trace Transformation Technique for Communication Refinement*, Proc. 9th International Workshop on HW/SW Codesign (CODES/CASHE'01), Copenhagen, Denmark, April 2001.

[62] P.Liverse, P.v.Wolf, E.Deprettere, K.Vissers, *A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems*, Proc. IEEE Workshop on Signal Processing Systems (SiPS'99), Taipei, Taiwan, 1999.

[63] H.Mooshofer, *Entwurfsmethodik für eine flexible Architektur zur Videoobjekt-Segmentierung*, Ph. Dissertation submitted to the Technische Universität München, 2002.

[64] K.Lahiri, A.Raghunathan, S.Dey *Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures*, Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD 99), San Jose, USA, November 1999.

[65] K.Lahiri, A.Raghunathan, S.Dey, *Performance Analysis of Systems With Multi-Channel Communication Architectures*, Proc. International Conference on VLSI Design, Calcutta, India, January 2000.

[66] K.Lahiri, A.Raghunathan, S.Dey, *System-Level Performance Analysis for Designing On-Chip Communication Architectures*, IEEE Transactions on Computer-Aided Design, vol.20, no.6, June 2001.

[67] F.Balarin, M.Chiodo, H.Hsieh, A.Jureska, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli, E.Sentovich, K.Suzuki, B.Tabbara, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers, 1997.

[68] J.Buck, S.Ha, E.A.Lee, D.D.Masserchmitt, *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, International Journal on Computing and Simulation, vol.4, pp.155-182, April 1994.

[69] L.Thiele, S.Chakraborty, M.Gries, S.Künzli, *A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures*, Proc. 39th Design Automation Conference (DAC'02), New Orleans, USA, June 2002.

[70] L.Thiele, S.Chakraborty, M.Gries, S.Künzli, *Design Space Exploration of Network Processor Architectures*, to appear as a book chapter in *Network Processor Design 2002: Design Principles and Practices*.

[71] L.Thiele, S.Chakraborty, M.Gries, A.Maxiaguine, J.Greutert, *Embedded Software in Network Processors - Models and Algorithms*, 1st Workshop on Embedded Software, Lake Tahoe, USA, 2001.

[72] L.Thiele, S.Chakraborty, M.Naedele, *Real Time Calculus for Scheduling Hard Real-Time Systems*, Proc. IEEE International Symposium on Circuits and Systems (ISCAS'00), vol.4, pp.101-104, Geneva, Switzerland, May 2000.

[73] J.P.M.Voeten, I.G.Stappers, M.C.W.Geilen, L.J.van Bokhoven, P.H.A.van der Putten, M.P.J.Stevens, *An Analytical Approach Towards System Level Performance Analysis*, Proc. IEEE/STW 10th Workshop on Circuits, Systems and Signal Processing, Utrecht, Netherlands, November 1999.

[74] J.P.M.Voeten, P.H.A.van der Putten, M.C.W.Geilen, M.P.J.Stevens, *Towards System Level Performance Modelling*, Proc. IEEE ProRISC'98, Utrecht, STW, Technology Foundation, pp.593-597, 1998.

[75] T.Yen, W.Wolf, *Performance Estimation for Real-Time Distributed Embedded Systems*, IEEE Transactions on Parallel and Distributed Systems, vol.9, no.11, November 1998.

[76] A.J.C.van Gemund, *Symbolic Performance Modeling of Parallel Systems*, IEEE Transactions on Parallel and Distributed Systems, January 2003.

[77] *The PAMELA Language*, http://ce.et.tudelft.nl/gemund/Pamela/pamela.html.

[78] A.C.J.Kienhuis, *Design Space Exploration of Stream-Based Dataflow Architectures: Methods and Tools*, Ph.Dissertation submitted to the Delft University of Technology, 1999.

[79] A.Baghdadi, N.E.Zergainoh, W.O.Cesario, A.A.Jerraya, *Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems*, IEEE Transactions on Software Engineering, vol.28, no.9, September 2002.

[80] P.Crowley, M.E.Fiuczynski, J.L.Baer, B.N.Bershad, *Characterizing Processor Architectures for Programmable Network Interfaces*, Proc. International Conference on Supercomputing, Santa Fe, USA, May 2000.

[81] L.Gwennap, B.Wheeler, *A Guide to Network Processors*, MicroDesign Resources, 1st edition, 2000.

[82] Douglas E. Comer, *Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture*, Prentice Hall, 1995.

[83] Andrew S. Tannenbaum, *Computer Networks*, Prentice Hall, 3rd edition, 1996.

[84] D.Husak, R. Gohn, *Network Processor Programming Models: The Key to Achieving Faster Time-to-Market and Extending Product Life*, C-Port White Paper.

[85] D.Husak, *Network Processors: A Definition and Comparison*, C-Port White Paper.

[86] M.Gries, *Algorithm-Architecture Trade-Offs in Network Processor Design*, Ph.Dissertation submitted to the Swiss Federal Institute of Technology Zurich (ETH), 2001.

[87] T.Wolf, M.A.Franklin, *COMMBECH - A Telecommunications Benchmark for Network Processors*, Proc. IEEE International Symposium on Performance Analysis of System and Software, Austin, USA, April 2000.

[88] T.Wolf, M.A.Franklin, E.Spitznagel, *Design Tradeoffs for Embedded Network Processors*, Internal Report WUCS-00-24, Washington University of St. Louis. July 2000.

[89] K.Coffman, A.Odlyzko, *The Size and Growth Rate of the Internet*, First Monday, http://www.firstmonday.dk/issues/issue3_10/coffman/index.html, 3, 10, October 1998.

[90] Corman, Leiserson, Rivest, *Introduction to Algorithms*, McGraw-Hill, 1002.

[91] C.P.Joshi, *A New Performance Evaluation Approach for System Level Design Space Exploration*, Master Thesis submitted to the Indian Institute of Technology Delhi, 2002.

[92] A.Rastogi, M.Balakrishnan, A.Kumar, *Integrating Communication Cost Estimation in Embedded Systems Design: A PCI Case Study*, Proc. of International Conference on VLSI Design, Bangalore, India, January 2001.

[93] P.Chou, R.Ortega, K.Hines, K.Partridge, G.Borriello, *The Chinook Hardware/Software Co-Synthesis System*, Proc. International Symposium on System Synthesis, Cannes, France, 1995.

[94] J.Henkel, R.Ernst, *High-Level Estimation Techniques for Usage in HW/SW Co-Design*, Proc. Asia South Pacific - Design Automation Conference (ASP-DAC'98), Yokohama, Japan, February 1998.

[95] M.Lajolo, A.Raghunathan, S.Dey, L.Lavagno, A.Sangiovanni Vincentelli, *A Case Study on Modeling Shared Memory Access Effects During Performance Analysis of HW/SW Systems*, Proc. 6th International Workshop on HW/SW Codesign (CODES/CASHE'98). Seattle, USA, March 1998.

[96] M.Abramowitz, I.A.Stegun (Eds.), *Stirling Numbers of the Second Kind*, 24.1.4 in *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*, 9th edition, New-York: Dover, pp.824-825, 1972.

[97] *The Combinatorial Object Server*, http://www.theory.csc.uvic.ca/cos/inf/setp/Set-Partitions.html.

[98] P.Eles, K.Kuchcinski, Z.Peng, A.Doboli, P.Pop, *Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems*, Proc. International Conference on Design Automation and Test in Europe (DATE'98), Paris, France, February 1998.

[99] N.Pazos, W.Brunnbauer, J.Foag, T.Wild, *System-Level Performance Estimation for VLSI Networking-Architectures Methodology*, Proc. International Workshop on IP_Based SoC Design, Grenoble, France, December 2001.

[100] N.Pazos, W.Brunnbauer, J.Foag, T.Wild, *System-Based Performance Estimation of Multi-Processing, Multi-Threading SoC Networking Architectures*, Proc. Forum on Specification and Design Languages (FDL'02), Marseille, France, September 2002.

[101] J.R.Levine, T.Mason, D.Brown, *lex & yacc*, O'Reilly & Associates, Inc. 2nd Edition, 1992.

[102] V.Paxson, *Flex, A Fast Scanner Generator, Version 2.5*, ftp://prep.ai.mit.edu/pub/gnu.

[103] C.Donnelly, R.Stallman , *Bison, The Yacc-Compatible Parser Generator, Version 1.25*, ftp://prep.ai.mit.edu/pub/gnu.

[104] J.Rowson, *Hardware/Software Co-Simulation*, Proc. IEEE Design Automation Conference (DAC'94), June 1994.

[105] L.Semeria, A.Ghosh, *Methodology for Hardware/Software Co-Verification in C/C++*, Proc. Asia South Pacific - Design Automation Conference (ASP-DAC'00), Yokohama, Japan, January 2000.

[106] *Intel IXP1200 Network Processor home page*, http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[107] F.Karim, A.Nguyen, S.Dey, R.Rao, *On-Chip Communication Architecture for OC-768 Network Processors*, Proc. 38rd Design Automation Conference (DAC'01), Las Vegas, USA, 2001.