HW-SW Components for Parallel Embedded Computing on NoC-based MPSoCs

Ph.D. Thesis in Computer Science
(Microelectronics and Electronics System Department)

Author:
Jaume Joven Murillo

Supervisor:
Prof. Jordi Carrabina Bordoll

Universitat Autonòma de Barcelona (UAB)

December 2009
Bellaterra, Spain

I certify that I have read this dissertation and that,
in my opinion, it is fully adequate in scope and quality
as a dissertation for the degree of Doctor of Philosophy.

Prof. Jordi Carrabina Bordoll

This work was carried out at Universitat Autonòma de Barcelona (UAB),
Ecole Polytechnique Fédérale de Lausanne (EPFL), and
ARM Ltd. R&D Department (Cambridge).

© 2009 Jaume Joven Murillo
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission from the copyright owner.

Science is an imaginative adventure of
the mind seeking truth in a world of mystery.

-Sir Cyril Herman Hinshelwood
(1897-1967) English chemist.
Nobel prize 1956.

Abstract

Recently, on the on-chip and embedded domain, we are witnessing the growing of the Multi-Processor System-on-Chip (MPSoC) era. Network-on-chip (NoCs) have been proposed to be a viable, efficient, scalable, predictable and flexible solution to interconnect IP blocks on a chip, or full-featured bus-based systems in order to create highly complex systems. Thus, the paradigm to high-performance embedded computing is arriving through high hardware parallelism and concurrent software stacks to achieve maximum system platform composability and flexibility using pre-designed IP cores. These are the emerging NoC-based MPSoCs architectures. However, as the number of IP cores on a single chip increases exponentially, many new challenges arise.

The first challenge is the design of a suitable hardware interconnection to provide adequate Quality of Service (QoS) ensuring certain bandwidth and latency bounds for inter-block communication, but at a minimal power and area costs. Due to the huge NoC design space, simulation and verification environments must be put in place to explore, validate and optimize many different NoC architectures.

The second target, nowadays a hot topic, is to provide efficient and flexible parallel programming models upon new generation of highly parallel NoC-based MPSoCs. Thus, it is mandatory the use of lightweight SW libraries which are able to exploit hardware features present on the execution platform. Using these software stacks and their associated APIs according to a specific parallel programming model will let software application designers to reuse and program parallel applications effortlessly at higher levels of abstraction.

Finally, to get an efficient overall system behaviour, a key research challenge is the design of suitable HW/SW interfaces. Specially, it is crucial in heterogeneous multiprocessor systems where parallel programming models and middleware functions must abstract the communication resources during high level specification of software applications.

Thus, the main goal of this dissertation is to enrich the emerging NoC-based MPSoCs by exploring and adding engineering and scientific contribution to new challenges appeared in the last years. This dissertation focuses on all of the above points:

- by describing an experimental environment to design NoC-based systems, xENoC, and a NoC design space exploration tool named NoC-Maker. This framework leads to a rapid prototyping and validation of NoC-based MPSoCs.
- by extending Network Interfaces (NIs) to handle heterogeneous traffic from different bus-based standards (e.g. AMBA, OCP-IP) in order to reuse and communicate a great variety off-the-shelf IP cores and software stacks in a transparent way from the user point of view.
- by providing runtime QoS features (best effort and guaranteed services) through NoC-level hardware components and software middleware routines.
- by exploring HW/SW interfaces and resource sharing when a Floating Point Unit (FPU) co-processor is interfaced on a NoC-based MPSoC.
- by porting parallel programming models, such as shared memory or message passing models on NoC-based MPSoCs. We present the implementation of an efficient lightweight parallel programming model based on Message Passing Interface (MPI), called on-chip Message Passing Interface (ocMPI). It enables the design of parallel distributed computing at task-level or function-level using explicit parallelism and synchronization methods between the cores integrated on the chip.
- by provide runtime application to packets QoS support on top of the OpenMP runtime library targeted for shared memory MPSoCs in order to boost or balance critical applications or threads during its execution.

The key challenges explored in this dissertation are formalized on HW-SW communication centric platform-based design methodology. Thus, the outcome of this work will be a robust cluster-on-chip platform for high-performance embedded computing, whereby hardware and software components can be reused at multiple levels of design abstraction.

Keywords

NETWORKS-ON-CHIP (NOCS)
MULTI-PROCESSOR SYSTEM-ON-CHIPS (MPSOCS)
NOC-BASED MPSOCS
ON-CHIP INTERCONNECT FABRIC
EDA/CAD NOC DESIGN FLOW
HW-SW INTERFACES
CYCLE-ACCURATE SIMULATION
QUALITY-OF-SERVICE (QOS)
MESSAGE PASSING
PARALLEL PROGRAMMING MODELS

Acknowledgments

A large number of people have been indispensable to succeed on the realization of this Ph.D. thesis.

First of all, I would like to thank my advisor Prof. Jordi Carrabina for their guidance and support over these years. Also, I would like to express my deeply gratitude to Prof. David Castells, Prof. Lluis Terés, and Dr. Federico Angiolini for their assistance and technical support provided, and for listening and helping me every time I needed it doing large research discussions and meetings.

Another special mention to Prof. Giovanni De Micheli, Prof. David Atienza and Prof. Luca Benini to open the door giving me the opportunity to do research at Ecole Polytechnique Fédérale de Lausanne (EPFL) side by side with many brilliant people (thanks to Ciprian, Srini, Antonio,...), as well as in Bologna (thanks Andrea).

And I am deeply grateful to Per Strid, John Penton, and Emre Ozer to give me the chance to live an incredible working experience in a world-class enterprise like ARM Ltd. at Research & Development Department in Cambridge, as well as all researchers at ARM Ltd R&D Department (specially to Derek, Akash and Antonino).

A special thanks to all the people that I had the pleasure to work with in Barcelona (all colleagues of CEPHIS Lab, specially mention to Sergi and Eduard).

Thought this Ph.D. I had the possibility to travel around Europe and meet very interesting people. However, special thanks to all people that I had the pleasure to meet during my Ph.D. research stays in Lausanne (Nestle ”Spanish mafia”, EPFL friends, ...) which help me to adapt to another country, and surely make me to live one of the best periods of my life.

Thanks to all my friends, even though they were not very convinced if I was working, studying, or both, during the last four years, they encourage me to chase the goal to finish the Ph.D.

Last but not least, I am deeply grateful to my family to support me unconditionally, and to educate me to do always as much as I can to obtain my targets. This dissertation is dedicated to them.

I cannot finish without acknowledge the institutions that supported me and my research with funding and infrastructure along these years. This Ph.D. dissertation has mainly supported by Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR) and Spanish Ministry for Industry, Tourism and Trade (MITyC) through TEC2008-03835/TEC and "ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (No 06015, June 2007 – May 2010), as well as, the collaboration grants awarded by European Network of Excellence on High Performance and Embedded Architecture and Compilation (HIPEAC).

Thanks to all of you...

...and now, in this "crisis" era, lots of fellows nowadays have a B.A., M.D., M.Ph, Ph.D, but unfortunately, they don't have a J.O.B

I hope this not happen to me , and I am expecting impatiently new positive opportunities to do what I like

Contents

Abstract i

Keywords iii

Acknowledgments v

Contents vii

List of Figures xi

List of Tables xv

Listings xvii

1 Introduction 1
- 1.1 Background . . . 5
- 1.2 NoC-based MPSoC: Opportunities and Challenges . . . 10
- 1.3 Contributions of This Dissertation . . . 12
- 1.4 Dissertation Outline . . . 17

2 Experimental Framework for EDA/CAD of NoC-based Systems 19
- 2.1 Networks-on-Chip: Preliminary Concepts . . . 19
  - 2.1.1 NoC Paradigm - General Overview . . . 20
  - 2.1.2 Topologies . . . 21
  - 2.1.3 Routing Protocol and Deadlock Freedom . . . 24
  - 2.1.4 Switching Communication Techniques . . . 26
  - 2.1.5 Buffering and Virtual Channels . . . 28
  - 2.1.6 Flow Control Protocols . . . 30
  - 2.1.7 Micro-network Protocol Stack for NoCs . . . 31
- 2.2 Motivation and Key Challenges . . . 34
- 2.3 Related work on NoC Design and EDA Tools . . . 35

2.4 xENoC environment - NoCMaker . . . 36
2.4.1 Design Flow of NoCMaker . . . 37
2.4.2 NoC modeling . . . 39
2.4.3 Metrics Extraction according to Design Space Points . . . 42
2.4.4 Modeling Traffic Patterns and Benchmarks . . . 48
2.4.5 System and Circuit Validation . . . 50
2.5 Experiments using xENoC – NoCMaker . . . 52
2.5.1 Synthesis of Nios II NoC-based MPSoC architectures . . . 53
2.6 Conclusion and Future Work . . . 55

3 Full-Featured AMBA AHB Network Interface OCP Compatible 57
3.1 General Concepts of ×pipes NoC library . . . 57
3.2 Motivation and Challenges . . . 58
3.3 AMBA & OCP-IP Interconnection Fabric Overview . . . 60
3.3.1 AHB/AXI Architecture Overview . . . 61
3.3.2 OCP-IP Overview . . . 64
3.4 Related work . . . 65
3.5 Full-Featured AMBA 2.0 AHB NI for ×pipes NoC . . . 67
3.5.1 Packet Definition and Extension . . . 68
3.5.2 AMBA 2.0 AHB Transactions to NoC Flits . . . 71
3.6 Experimental results . . . 78
3.6.1 Functional Verification on a AMBA-OCP Start NoC . . . 78
3.6.2 Synthesis of AMBA 2.0 AHB NI vs. OCP NI . . . 81
3.7 Conclusion and Future Work . . . 83

4 Runtime QoS support Through HW and Middleware Routines 85
4.1 Motivation and Challenges . . . 85
4.2 Related work . . . 87
4.3 NoC-level QoS Design . . . 89
4.3.1 Trigger QoS at Network Interfaces . . . 90
4.3.2 QoS Extensions at Switch Level . . . 93
4.4 Low-level QoS-support API and Middleware Routines . . . 96
4.5 QoS Verfication and Traffic Analysis . . . 98
4.6 Synthesis results . . . 100
4.6.1 QoS Overhead at Network Interface . . . 101
4.6.2 QoS Overhead at Switch Level . . . 102
4.7 Conclusion and Future Directions . . . 105

5 HW-SW FPU Accelerator Design on a Cortex-M1 Cluster-on-Chip 107
5.1 Motivation and Challenges . . . 107
5.2 Related Work . . . 109

5.3 Cortex-M1 FPU Accelerator – HW Design 111
5.4 HW-SW CPU-FPU Communication Protocol 116
5.5 FPU Hardware-Dependent Software Library Extensions 119
5.6 Experimental results 122
5.6.1 Cortex-M1 SP FPU Hardware Synthesis 122
5.6.2 Performance Studies – HW-FPU vs. SW emulation . . 126
5.7 Conclusion and Future Work 130

6 Parallel Programming Models on NoC-based MPSoCs 133
6.1 A General Overview of Parallelism 133
6.2 Traditional Parallel Programming Models 137
6.3 Motivation and Key Challenges 139
6.4 Related work 141
6.5 Embedding MPI for NoC-based MPSoCs 146
6.5.1 Message Passing Interface (MPI) - General Concepts . 147
6.5.2 MPI Adaptation Towards NoC-based MPSoCs 150
6.5.3 Encapsulation, and on-chip MPI Packet Format 151
6.5.4 A Lightweight on-chip MPI Microtask Stack 151
6.5.5 Software Stack Configurations 153
6.5.6 Message Passing Communication Protocols 154
6.5.7 Improving ocMPI Through Tightly-Coupled NIs 155
6.6 OpenMP Framework for NoC-based MPSoCs 157
6.7 Evaluation of on-chip Message Passing Framework 160
6.7.1 Profiling the Management ocMPI Functions 160
6.7.2 Low-level Message Passing Benchmarks 160
6.7.3 Design of Parallel Applications using ocMPI library . . 164
6.8 Improve OpenMP Kernels using QoS at NoC-level 166
6.8.1 Balancing and Boosting Threads 167
6.8.2 Communication Dominated Loop 169
6.8.3 Computation Dominated Loop 170
6.8.4 Balance Single Thread Interference 170
6.9 Conclusion and Open Issues 171

7 Conclusions and Future Work 175
7.1 Overview and General Conclusions 175
7.2 Open Issues and Future Work 180

A Author's Relevant Publications 183

B Curriculum Vitae 185

Glossary 187

Bibliography 193

List of Figures

1.1 HW-SW evolution of the productivity gap 2
1.2 Evolution of microelectronics, from ASICs to NoCs in SoCs . . 3
1.3 Evolution of the communication fabrics in MPSoCs 6
1.4 HW-SW communication-centric platform-based design 14

2.1 Logical view of a NoC-based system 21
2.2 Classification of NoC design choices 21
2.3 Representative subset of NoC topologies 23
2.4 Classification of NoC systems (granularity vs. homogeneity) . 24
2.5 Turn model for deadlock-free adaptive routing 25
2.6 Packet transmission using different switching techniques . . . 28
2.7 Micro-network stack for NoC-based system 33
2.8 State of the art of academic EDA NoC Tools 36
2.9 Design space exploration – design effort vs. flexibility 37
2.10 NoCMaker design flow (synthesis and metrics extraction) . . 39
2.11 Definition of a NoC design point in NoCMaker 41
2.12 Point-to-point statistics and traffic analysis in NoCMaker . . . 44
2.13 Interactive simulation window in NoCMaker 44
2.14 Area usage window in NoCMaker 46
2.15 Overall power usage window statistics in NoCMaker 48
2.16 Custom/pre-defined synthetic traffic patterns in NoCMaker . 50
2.17 Verification of NoC-based systems using NoCMaker 51
2.18 Packet sequence analysis window in NoCMaker 52
2.19 Nios II NoC-based MPSoC architecture 53
2.20 Circuit switching vs. ephemeral circuit switching 53
2.21 Synthesis of Nios II NoC-based MPSoCs 54
2.22 Area breakdown of our Nios II NoC-based MPSoC 55

3.1 NI overview on a multi-protocol NoC-based system 60
3.2 Block diagram of AMBA 2.0 AHB (bus vs. multi-layer) 62
3.3 AXI Overview, and AXI vs. AMBA 2.0 AHB transactions . . . 63

3.4 AHB NI block diagram in ×pipes NoC 68
3.5 AMBA 2.0 AHB – OCP request packet 68
3.6 AMBA 2.0 AHB – OCP response packet 70
3.7 AMBA 2.0 AHB NI request FSM 73
3.8 AMBA 2.0 AHB NI receive FSM 75
3.9 AMBA 2.0 AHB NI resend FSM 76
3.10 AMBA 2.0 AHB NI response FSM 77
3.11 Single switch AMBA 2.0 AHB – OCP star topology 79
3.12 Synthesis of AMBA 2.0 AHB NI vs. OCP NI 82
3.13 Circuit performance (f_{max}), AMBA 2.0 AHB vs. OCP NIs . . . 83

4.1 QoS extension on the header packets 92
4.2 Architecture overview of ×pipes switch 94
4.3 Allocator block diagram & arbitration tree 95
4.4 Topology 5x5 clusters NoC-based MPSoC 99
4.5 Validation and traffic priorization under different scenarios . . 100
4.6 AMBA – OCP Network Interfaces (QoS vs. noQoS) 101
4.7 QoS impact (LUTs, f_{max}) at NI level (AMBA-OCP) 102
4.8 Switch synthesis varying the arity (QoS vs. noQoS) 103
4.9 QoS impact (f_{max}, LUTs) at switch level 104
4.10 QoS impact (LUTs, f_{max}) according to the priority levels 105

5.1 Single and double precision IEEE 754 representation 109
5.2 Cortex-R4 Floating Point Unit (FPU) 112
5.3 Cortex-M1 AHB SP memory mapped FPU accelerator 113
5.4 SP FPU accelerator memory mapped register bank 113
5.5 Floating Point OpCode register (FP Opcode) layout 114
5.6 Floating Point System ID register (FPSID) layout 114
5.7 Floating Point Status and Control register (FPSCR) layout . . . 115
5.8 Floating Point Exception Register (FPEX) layout 115
5.9 CPU – FPU hardware-software protocol 117
5.10 Cortex-M1 MPSoC cluster architecture 124
5.11 FPU execution times using different CPU-FPU notifications . . 127
5.12 Cortex-M1 SP FPU speedup vs. SW emulation FP library . . . 128
5.13 FPU execution times on Cortex-M1 MPSoC 129
5.14 HW FPU speedup vs. SW emulation on Cortex-M1 MPSoC . . 129
5.15 HW-SW interface co-design on embedded SoC architectures . 131

6.1 Multiple levels and ways of parallelism 140
6.2 On-chip message passing interface libraries 146
6.3 Logical view of communicators and messages in MPI 147

6.4 Patterns of the MPI collective communication routines 149
6.5 MPI adaptation for NoC-based systems 150
6.6 onMPI/j2eMPI message layout 151
6.7 Different software stack configurations (ocMPI library) 153
6.8 Eager vs. rendezvous message passing communication 154
6.9 Tight-coupled NI block diagram 156
6.10 HW-SW view of our OpenMP platform 159
6.11 Profiling of the management ocMPI functions 160
6.12 Comparison of a bus-based vs. tight coupled NI 161
6.13 Execution time of `ocMPI_Init()` & `ocMPI_Barrier()` . . . 162
6.14 Execution time of point-to-point and broadcast in ocMPI . . . 163
6.15 Parallelization of cpi computation 165
6.16 Parallelization of Mandelbrot set application 166
6.17 QoS applied to unbalanced OpenMP programs 169
6.18 Balance unbalanced OpenMP programs using QoS 171

7.1 SoC consumer design complexity trends 176
7.2 Thesis overview and NoC design space overview 177
7.3 HW-SW design costs on future SoC platforms 181

List of Tables

1.1 Bus-based vs. on-chip network architectures 7
1.2 State of the art in MPSoCs 8
1.3 NoC research topics according to the abstraction levels 11

2.1 Buffering costs for different packet switching protocols 29

3.1 Mapping AHB – OCP signals on the request header 69

5.1 Cortex-M1 SP FPU accelerator core latencies 118
5.2 Synthesis of single and double precision PowerPC FPU 123
5.3 Cortex-M1 SP FPU synthesis on FPGA devices 123
5.4 Synthesis of Cortex-M1 MPSoC cluster without FPU 126
5.5 Synthesis of Cortex-M1 MPSoC cluster with FPU 126

6.1 Parallel architectures according to memory models 136
6.2 Supported functions in our on-chip MPI libraries 152
6.3 cpi and Mandelbrot set problem size parameters 165
6.4 Parameters of our OpenMP NoC-based MPSoC 168

Listings

2.1 XML example of a NDSP in NoCMaker 38
3.1 AMBA-OCP topology description file 80
4.1 QoS encoding on header packets 90
4.2 Low-level middleware NoC services API 96
4.3 QoS middleware functions 97
5.1 CPU – FPU protocol definition 116
5.2 *FSUB* implementation using a HW FPU accelerator 120
5.3 HW FPU accelerator software routines 121
5.4 Floating point software emulated routines 121
6.1 Tight-coupled NI instructions 157

CHAPTER 1

Introduction

Nowadays, the continuous evolution of the technology (i.e. Moore's law, that claims that every 18 months the number of transistor on a chip is duplicated) causes that every INTEGRATED CIRCUIT is able to contain a large number of transistors and this will continuously grow until 2020 according to the *The International Technology Roadmap for Semiconductors* prediction [1]. Assuming this prediction the designers must change the methodologies and also the associated EDA tools to exploit all the potential of the technology effectively. Furthermore, from 2005 until now, due to the importance of software components (i.e. SW stacks, libraries and stand-alone components) that full-featured systems require, the productivity gap includes the additional software required to the hardware platform. This software is crucial to enable the flexibility and programmability by exploiting all hardware available features. In addition, to boost microelectronics productivity and reduce specially the hardware gap between technology and design productivity, the designers must change their mind at the moment of planning the chip design process. Figure 1.1 on the following page shows the evolution of the productivity gap during the last years.

Because of this law, during the last 30 years until our days the microelectronics has experimented a continuous and a natural rapid evolution in many aspects, such as:

- Design methodologies (i.e. bottom-up, top-down, meet-in-the-middle, communication-centric...).
- COMPUTER AIDED DESIGN (CAD)
ELECTRONIC DESIGN AUTOMATION (EDA) tools.
- Abstraction levels of design (i.e. layout, schematic, system-level...).

- Technology integration (i.e. LSI, VLSI,...).

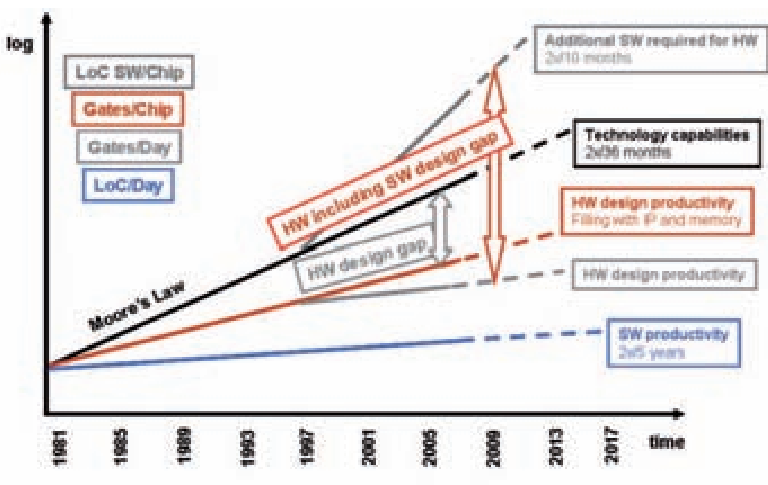


Figure 1.1: HW-SW evolution of the productivity gap

As we shown in Figure 1.2 on the next page, in 70s when microelectronics saw the light, the designs were based on a *Bottom-Up* methodology which consists in using the layout design rules and transistor models or, later, a library of basic cells, to implement the functionality of your IC. During this decade, EDA tools let the designers to create the schematic and the layout of their IC quickly, but designers were not able to simulate the whole IC until it was completely developed. In addition, designers were obliged to select a particular technology, and as a consequence, they also got the corresponding library of basic cells. Thus, a design error will probably be detected at the block composition level, and this will cause a big problem during last phases of design. Therefore, to avoid these problems of portability and the lack of a global verifiable view, the emergence of *Hardware Description Language (HDL)* took place, changing this methodology to a top-down approach. Using a *Top-Down* methodology, designers were able to specify all requirements at first phases of design in order to verify functional requirements, to later implement the circuit using a synthesis tools based on portable hardware description languages (technology-independent HDL), map the circuit to a specific technology, and after that check if the IC fulfills detailed performance requirements (using back-annotation). If it does not, designers had to do a feedback phase in order to improve the design (changing the design or selecting another technology) that fulfills the requirements, this is a *spiral model* [2].

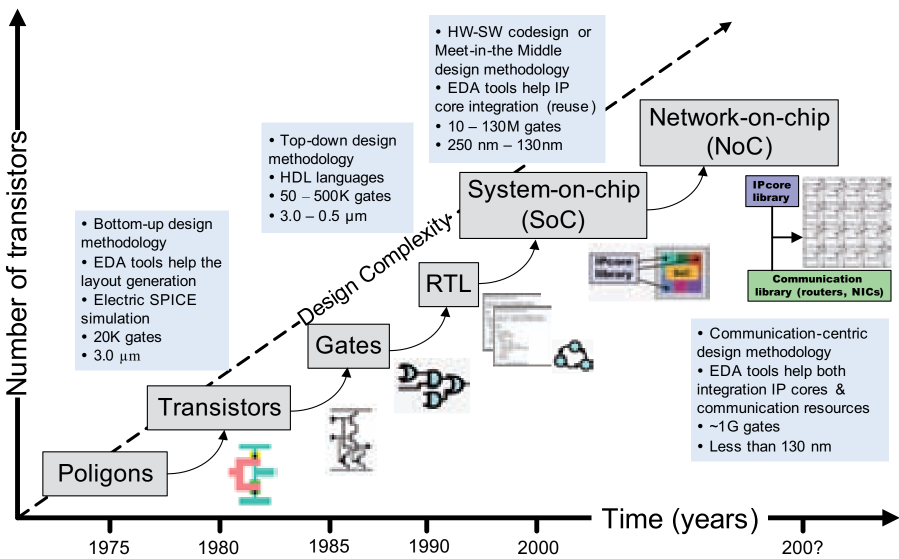


Figure 1.2: Evolution of microelectronics, from ASICs to NoCs in SoCs

Furthermore, through HDLs let designers to move towards higher levels of abstraction due to a design certification (sign-off) based on cell library characterization. So, for most of digital design, it was not necessary to design at transistor level because using HDLs, designers were able to design above *Register Transfer Level (RTL)* descriptions. Designing at RTL level made possible to design large chips within shorter design cycles. Furthermore, this RTL design methodology at the present is still a de facto standard for making large cell-based ASICs quickly.

Despite of RTL design advantages, during 90s designers realized that the existing methodologies were not enough to reduce the productivity gap, the design costs and the time-to-market. Therefore, designers decided to set up strategies to reuse all previously implemented and tested RTL designs, named as *Intellectual Property (IP cores)*. Thus, the reuse of these IP cores together with the HW-SW co-design strategies, and the improvement of EDA tools were the solution adopted to face part of that problem by boosting the productivity. Thus, more complex IP could be designed, from layouts containing special transistor structures, until functional blocks, such as ALUs and multipliers, to microprocessors, ASIPs, VLIW, DSPs (cores and surrounding infrastructure) and also verification IPs, which they have become the primitive building blocks when designers implement an IC. For these reasons, designers are always moving towards higher levels of abstraction, describing the system's functionality using these components and its related methodologies. Thus, designers will be able to realize complex

IC reducing the design time (and therefore the time to market).

However, the increase of design productivity was not enough to reduce the gap and scaling technological problems appear. *Deep Submicrometer (DSM)* and *Sub-wave Length (SWL)* technology effects and interconnection complications appear when designers develop complex ICs with many IP cores. The most common submicron effects are crosscoupling, noise and transient errors, and a global clock synchrony. The clock signal will soon need several clock cycles to traverse the chip, clock skew become unmanageable, and the clock distribution tree is already today a major source of power consumption and area cost. In addition, with higher clock frequencies these problems became more significant.

On the other hand, the interconnection problems are also a huge trouble, and it is because of technology scaling works better for transistors than for interconnection wires [3] (though the number of metal layers is also increasing). This leads gradually to its dominant contribution in performance figures, power consumption and speed. All these reasons carry us to the obvious conclusion that we need something more than only reuse IP-cores in order to develop complex ICs with billion of transistors in the near future.

Next step forward in design methodologies was to move from bottom-up or top-down methodologies towards a hybrid or combination of both, this resultant methodology is called *Meet-in-the-Middle*. It leverages the power of top-down methods and the efficiency of bottom-up styles. Using this methodology and due to the HW-SW co-design, let designers to do concurrently the development of HW and SW components, and also select a HW-SW interface infrastructure at the moment of design a SoC. In addition, this methodology balances the design time with the implementation automation to get first-time-right designs. From this meet-in-the-middle approach, a more advanced and generic methodology that abstract architectural concepts emerged around year 2000, as the platform-based design [4,5]. This methodology is also based on reuse of integration of HW and SW components, and furthermore it is an extension of their predecessors because claim on reuse of system architectures and topologies. Moreover, this methodology also defines several independent levels, such as virtual and real platforms. Finally, it also includes domain specific software layers that helps in application development and the industrialization layer that helps to reduce costs.

To achieve all these trends, an emerging paradigm called SYSTEM-ON-CHIP (SOC) or MULTI-PROCESSOR-SYSTEMS-ON-CHIP (MPSOCS) and its associated concepts appear in order to design the new generation of IC on the embedded domain.

MPSoCs [6,7] are used in many different applications and environments, such as multimedia mobile (e.g. mobile computing, video and audio coding, photography), robotics and automation (e.g. virtual reality), SDR communications (e.g. GPS, 3G, UMTS, HSDPA), networking, health care devices, stand-alone aerospace and automotive, an many more. Thus, the requirements, in terms of IP cores, that will require these systems will be between tens or hundreds of IP cores on a single die. The trend is that every 2 years the number of IP cores are doubled on a single chip [8].

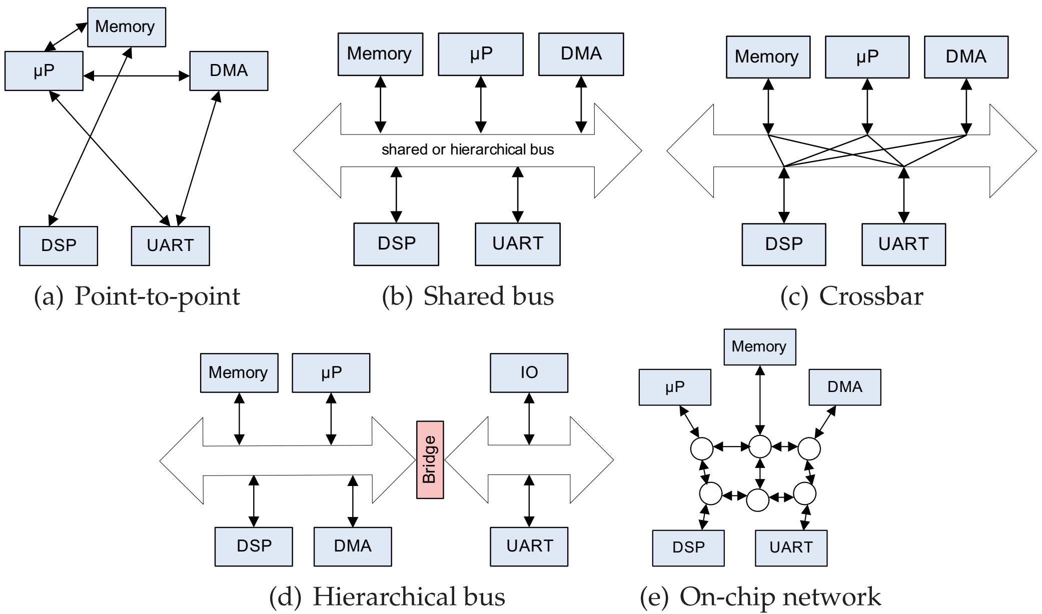
Depending of the system composability and the traffic diversity, we can talk about homogeneous (e.g. *Symmetric multiprocessing (SMP)* systems) or heterogeneous (e.g. *Asymmetric multiprocessing (ASMP)* systems), and general-purpose or application specific MPSoCs. Homogeneous MPSoCs are composed by many identical cores (e.g. using exclusively replications of microprocessor) provide specialized processing around a main processor, which allows for performance, power efficiency, and high reuse. On the other side, heterogeneous MPSoCs integrate specialized cores (e.g. DSPs, VLIWs with microprocessors and specialized HW blocks) around the main processor which not identical from the programming and implementation point of view. For many applications, heterogeneous AMP is the right approach, mainly due to power efficiency, but still homogeneous SMPs are attractive when the applications cannot be predicted in advance.

1.1 Background

A critical piece of MPSoC is the communication infrastructure, specially when the number of IP cores integrated on the system increase. The communication fabrics of these IP cores have been organized in different architectures during MPSoC era (see Figure 1.3 on the following page).

In the beginning due to the simplicity of the designs and traffic uniformity, MPSoCs were using a *Point-to-Point* based communication scheme [9] (see Figure 1.3(a) on the next page). In this architecture, all IP cores were able to communicate with each other (when required) only by using dedicated wires. The communication infrastructure is fixed between all IP cores rather than reusable and flexible. Despite this fact, a point-to-point architecture is optimal in terms of bandwidth availability, latency and power usage. Normally this architecture is used to design specific purpose IC.

Once IP cores were established in the market by third parties a big problem cropped up: how designers could interconnect these IP cores? Assuming that a complete design can integrate many IP cores, and each IP core might have different interfaces because it can come from different partners



(a) Point-to-point (b) Shared bus (c) Crossbar

(d) Hierarchical bus (e) On-chip network

Figure 1.3: Evolution of the communication fabrics in MPSoCs

or third parties. So, two approaches emerged to standardize this interconnection, one is known as VSIA [10, 11] with their own concept of *Virtual Component Interface (VCI)*, and the other is OCP-IP [12]. Both included a definition of a standard communication socket, allowing that some IP cores from different vendors could be plugged effortlessly together, and also a set of architectural rules (e.g. separated address and data buses). These rules, mainly VSIA, were widely adopted for the different interconnection structures, principally related to busses, of the processor core producers, in the processor-centric dominant approach, whereas OCP-IP was better addressed for a communication centric approach. Thus, the design of complex MPSoCs relies on the existence of EDA tools for design space exploration to help designers to integrate the cores on the system effortlessly.

In consequence, to get more flexibility, a next step forward was to change this point-to-point architecture to a *Shared Bus* infrastructure as shown in Figure 1.3(b), which is often called *On-Chip Bus (OCB)*. On it all IP cores (i.e. CPUs, DMA, memory controllers,...) are connected through the same backbone, i.e. sharing the communication resources, and devices are memory mapped on the address space of the bus, which lets to a global component view of the whole system. Since the introduction of a bus-based SoC architecture at the end of 90s, the solution to implement the SoC communication has generally been characterized by custom ad-hoc design mixing

buses and point-to-point links. These kind of architectures are built around well understood concepts, and they are extremely simple and easy to design. However, for the next generation of complex MPSoCs with many functional units, and traffic loads, this shared backbone quickly becomes a communication bottleneck, since only a pair of devices can communicate at the same time. In other words, as more units are added to the bus, its usage per communication event grows leading to higher contention and bandwidth limitations. The longer wires due to the increasing number of components along the chip and its associated capacitive yield also to a severe scalability problem, and a dramatically reduction of the overall operation frequency of the system. This fact is even worse when the technology scales down. Finally, for multi-master busses, the arbitration policy (e.g. fixed priorities, round robin, or time slot access) problem is also not trivial.

Bus Pros & Cons		Network Pros & Cons	
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth	X	Only point-to-point one-way wires are used for all network sizes, thus local performance is not degraded when scaling	✓
Bus timing is difficult in a DSM process	X	Routing decisions are distributed, if the network protocols is made non-central	✓
The bus arbiter is instance specific	X	The same router my be reinstantiated for all network sizes (communication reusability)	✓
Bus testability is problematic and slow	X	Locally placed dedicated BIST is fast and offers good test coverage	
Bandwidth is limited and shared by all units	X	Aggregated bandwidth scales with the network size	✓
Bus latency is wire-speed once arbiter has granted the control	✓	Internal network contention may cause large latencies	X
Any bus is almost directly compatible with most available IPs, including software running on CPUs	✓	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems	X
The concepts are simple and well understood by designers	✓	System designers need re-education for new concepts	X

Table 1.1: Bus-based vs. on-chip network architectures (adapted from [13])

Along this bus-based era, many standard bus-based SoCs have been developed by silicon companies. The most extended pairs OCB and embedded processor cores to build SoCs or MPSoCs are the followings:

- AMBA 2.0 [14, 15] $\leftrightarrow$ ARM or LEON3 [16].
- IBM Coreconnect [17] $\leftrightarrow$ PowerPC/Microblaze [18].
- Avalon [19] $\leftrightarrow$ Nios II [20]

In consequence, to face these limitations, pure shared bus-based systems shift towards a *Hierarchical* [14, 17] or *Multi-Layer* [27] global commu-

MPSoC	Technical Components	Architecture overview
Philips Nexperia ™ [21]	- MIPS PR3940, Trimedia TM32 VLIW - Three buses: Memory, MIPS and Trimedia - 2D rendering & MPEG-2 video decoder, image & composition, video input processor, scaler and system processor - I/O: UART, PCI, IEEE 1394, etc - OS: Windows CE, Linux, WindRiver, VwWorks - Use semaphores to access shared resources - Use in multimedia applications, digital video systems, DTV and set-top box - First experimental family was Viper PNX-8500 [22]	
TI OMAP ™ [23]	- ARM7, MCU and TMS320C54x for GPRS/GSM & ARM926EJ-S - 2D Graphic accelerator, shared memory, frame buffer, cryptography engines - Support all leading OS - I/O: bluetooth, USB, UART, irDA, WLAN - Support 2.5G and 3G mobiel applications - Security speed processing, gaming and multimedia mobile	
ST Nomadik ™ [24]	- Heterogeneous MPSoC (ARM926E-JS) AMBA bus and video/audio accelerators (MMDSP+ ST CPU) - I/O: UART, USB, Bluetooth, WLAN, GPS, FIrDA - 20 mW decoding MPEG-4 at 15 fps using QCIF + MP3 audio - I/O: bluetooth, USB, UART, irDA, WLAN - Supports OMAP architecture - Security speed processing, gaming and mobile multimedia applications (MPEG-4, H263)	
Cell Processor [25]	- Heterogeneous MPSoC (PE+PPE) PPE is 64-bits dual-threaded mP, PowerPC 8 SPE (SIMD processors 128 x 128 bit registers - Element Interconnect Bus - Support VMX - Consumes 60-80W@4GHz (256 GFLOPS SP and 25 GFLOPS DP floating point operations) - Used in 3D games consoles - Cryptography, graphics transform and lighting, physics, FFTs, matrix operations	

Table 1.2: State of the art in MPSoCs (extracted from [26])

nication architectures. In these architectures, multiple buses are interconnected through *bridges* (see Figure 1.3(d) on page 6) or *full or partial crossbars* (see Figure 1.3(c) on page 6). This evolution makes long wires shorter due to its segmentation to avoid signal degradation, and busses are implemented as multiplexed structures to reduce capacitance (and therefore power usage) and increase responsiveness (see Table 1.1 on the previous page). Many MPSoCs have been manufactured by different companies targeted to embedded platform domain. Table 1.2 presents a summary of the most extended MPSoC platforms targeted to multimedia, networking and wireless mobile applications.

In response to these issues, more advanced busses [28–30], advanced multi-layer communication infrastructures [31] or IP socket based protocols, such as AMBA-AXI [32] or OCP-IP [12] have been developed to mainly generate more flexible topologies, smarter distributed arbitration schemes, and more sophisticated handshake protocols (i.e. out-of-order transactions, multiple burst types, pipeline transactions, etc). However, these approaches are still sub-optimal due to the inherent bus limitation issues explained before in Table 1.1 on page 7, and are definitely not suitable for the large future scalable next-generation MPSoCs as is reported in [33–38].

Therefore, a natural evolution and its associated paradigm from the original shared, hierarchical bus-based architectures has been proposed by the research community to face the design of the next-generation of SoCs and MPSoCs.

This paradigm is built upon the NETWORKS-ON-CHIP (NOCS) concept [3,13,39–42]. We view a SoC as a micro-network of components, where the network is the abstraction of the communication among components and must satisfy many constraints (e.g. Quality-of-Service requirement, reliability, performance and energy bound, etc). Using this on-chip network communication architecture, the throughput is increased by pipelining the point-to-point wires, but also having parallel channels since the bus-bridges become distributed routing nodes.

As a replacement for busses and point-to-point links, NoCs hold the potential of scalability from ground up, modular and flexible design flow, as well as they address fundamental physical-level problems introduced when scaling microchip technologies into DSM geometries. In addition, NoC greatly improve topologically the scalability of SoCs, and shows higher power efficiency in complex SoCs compared to buses.

In the research community, there are two widely held perceptions of NoCs:

- NoC as a subset of SoC.
- NoC as an extension of SoC.

In the first view, the NoC is defined strictly as the data forwarding communication fabric, i.e. the network and methods used in accessing the network. In the second view, the NoC is defined more broadly (like a composition of SoCs), also to encompass issues dealing with the application, system architecture, and its impact on communication or vice versa.

In contrast to traditional SoC communication schemes outlined above in Table 1.2 on the facing page, the distributed of segmented scheme used in NoCs with specific communication links and routing nodes, scale well

in terms of chip size and complexity. A priori, using NoCs will be able integrate in a single chip tens to hundred cores without losing throughput. An additional advantage of this on-chip network architecture is that it is able to increase aggregated performance by exploiting parallel operations; it is an very important issue on MPSoCs.

To solve the distribution and the global clock synchrony, a *Globally Asynchronous Locally Synchronous (GALS)* scheme have been proposed to design NoC systems. Using this scheme it is not necessary to use a global clock for all the NoC. Each tile can have an independent clock, and the global synchronization is implemented using an asynchronous style. Therefore, it is not necessary a global chip clock synchrony and the clock tree complexity is reduced. Thus, the GALS scheme avoids increasing clock skew problems improving notably the power consumption.

Despite this large number of presented methodologies in this chapter, now MPSoCs designs are progressively requiring a *Communication-Centric and System-Level Platform-Based design* methodology. In this methodology let designers to reuse both, HW and SW components, but also the most important innovation; designers are able to reuse the communication architecture.

In terms of reusability, NoCs let the designers to reuse both, the computation nodes (using pre-designed IP cores) and the communication infrastructure (reusing routers/switches and network interfaces (NIs) or hardware wrappers), being crucial to reduce the productivity gap presented in Figure 1.1 on page 2.

In conclusion, NoCs constitute a viable solution paradigm to communicate SoCs, or future scalable large MPSoCs. Thus, the trend in a near future will be the custom design of NOC-BASED MPSOC architectures.

1.2 NoC-based MPSoC: Opportunities and Challenges

The term NoC is used in research today in a very broad sense, ranging from gate-level physical implementation, across system layout aspects and applications, to design methodologies and tools (see Table 1.3 on the facing page).

Hence, many opportunities and challenges arise specially at hardware design, but once the hardware design is fully functional and optimized lots of research topics emerge, such as parallel programming models, HW-SW interfaces, application mapping, run-time reconfiguration, O/S support, and multiple memory hierarchy options.

Abstraction Level	Research Topics
System/ Application Level	- Design methodologies, co-exploration, modeling, simulation, EDA tools - Architecture domain: mapping of applications onto NoCs, system composition, clustering, reconfigurability, Chip-multiprocessor, SMPs, or ASMPs NoC-based systems - NoC benchmarks - Traffic modeling: latency-critical, data-stream and best-effort - Parallel programming models - High-Level languages to exploit concurrency and parallelism - O/S support for NoC - Power and thermal management - Dynamic Voltage and Frequency Scaling (DFVS) - Run-time services according NoC features
Network Interfaces	- Functionality: encapsulation, service management - Sockets: plug and play, IP reuse
Network Architecture	- Topology: regular vs. irregular topologies, switch layout - Protocols: routing, switching, control schemes, arbitration - Flow control: deadlock avoidance, virtual channels, buffering - Quality of Service: service classification and negotiation - High-Level languages to exploit concurrency and parallelism - Features: error-correction, fault tolerance, broadcast/multicast/narrowcast, virtual wires
Link Level	- Timing, synchronous/asynchronous, reliability, encoding - Physical design of interconnect - Signaling and circuit design for links - Quality of Service: service classification and negotiation - NoCs for FPGAs and structured ASICs - 3D NoC Integration

Table 1.3: NoC research topics according to the abstraction levels

There are several key design questions that must be answered in developing a NoC-based MPSoCs for specific next-generation embedded applications:

- Extraction of concurrency from the application software or models decades of research into trying to automate the discovery of concurrency in software, but very little in the way of generally applicable results have emerged. Therefore, the integration of NoC architectures and with parallel programming models is the next big challenge.

- Homogeneity and Heterogeneity - For many applications, only an heterogeneous AMP approach is valid, but homogeneous SMP is also attractive when the applications cannot be predicted in advance. How can we combine both approaches where needed? How do the two domains communicate and synchronize?

- How many processors are required for an application, and are they best designed or configured as the composition of discrete subsystems or as the melding of them to share resources?

- What kind of on-chip communications architectures are best? Does NoC have wide applicability? How do we effectively mix NoC, buses, and point to point approaches?

- Scalability: It is relatively easy to visualize how to get to 10, 20, and 100 processors in a complex SoC composing many multiprocessor subsystems. How do you get to 500 to 1000 or more? Do we even want to? What are the practical limits?

In this dissertation we will try to answer part of these general questions. The novelty and contributions are extracted from the integration of different ideas on different research areas rather than exploiting specific research topics. Thus, applying well-know concepts and adapting them to the existing requirements will lead to a novel and reusable component approach to create a a working framework for high-performance embedded computing on on the emerging parallel NoC-based MPSoCs architecturesd. In Section 1.3 will show the main contribution of this Ph.D. dissertation.

1.3 Contributions of This Dissertation

This dissertation aims to shed light on the trade-offs in the design of NoC-based MPSoCs systems at many different levels, ranging from specific NoC-level features until its integration and programmability through high-level parallel programming models at application level. This vertical approach also takes into a count the design of efficient hardware-software interfaces, and the inclusion on the system of full-featured cores to achieve high-performance embedded computing. The main contributions are:

- The contribution of a NoC component library for simulation and synthesis called NoCMaker within xENoC environment. This library is highly configurable and targeted to evaluate the interconnection of different homogeneous and heterogeneous NoC-based MPSoCs.

- The analysis of NoC performance and cost in a variety of environments and traffic conditions as well as in real parallel applications. Results include architectural simulations and back-end analysis on operating frequency, area and power consumption.

- The extension of NoCs with facilities able to guarantee interoperability of IP core compliant with OCP-IP and AMBA 2.0 standard. This is a key features in today's system design in order to enable the reuse

and the transparent integration of different types of IP cores on a full-featured NoC-based MPSoC.

- The design and the extension of NoC components to guarantee different levels of Quality-of-Service (QoS). The main target is to provide a combination of guaranteed services and best-effort QoS. The QoS is provisioned by hardware and middleware components in order to enable run-time QoS reconfiguration at application level.

- The integration of all NoC-level blocks and SW libraries in a cycle-accurate simulation and traffic generation environment for validation, characterization and optimization purposes.

- HW-SW integration of Floating-point Unit (FPU) on a NoC-based MPSoC. This work will explore the hardware and software issues when interfacing a FPU on a MPSoC system, such as synchronization, resource sharing, HW-SW communication interfaces and protocols, etc. Results show the trade-offs between a pure HW block, and the SW emulation during the execution of floating point operations.

- The implementation of an efficient lightweight message passing programming model, inspired in MPI (Message Passing Interface), called on-chip MPI (ocMPI). It has been integrated and tested successfully in a cycle accurate simulator, but it also has been ported to a real FPGA-based NoC-based MPSoC to run parallel application and benchmarks.

- The study of a vertical integrated approach on how to enable QoS at NoC-level from application level using middleware routines. The study is focus on provide QoS at task and thread level to balance the workload under congestion, to boost performance of critical threads on OpenMP programs, and in general to meet application requirement.

Moreover, as a contribution to the field on the design of NoC-based systems, in this thesis is provided a structured overview of the state-of-the art in each chapter according to the specific research topic. Furthermore, as a proof of the concepts, all theoretical contributions have been deployed in a practical implementation on top of cycle accurate simulators and/or real prototyping platforms in order to obtain reliable accurate experimental results.

This dissertation also contributes on the formal definition of a HW-SW design methodology presented in Figure 1.4 on the following page to design highly parallel NoC-based MPSoCs for high-performance embedded

computing. The proposed methodology will be used along this thesis, and it is sustained on top of the definition of a robust OSI-like layered micro-network protocol stack adapted to NoC-based systems.

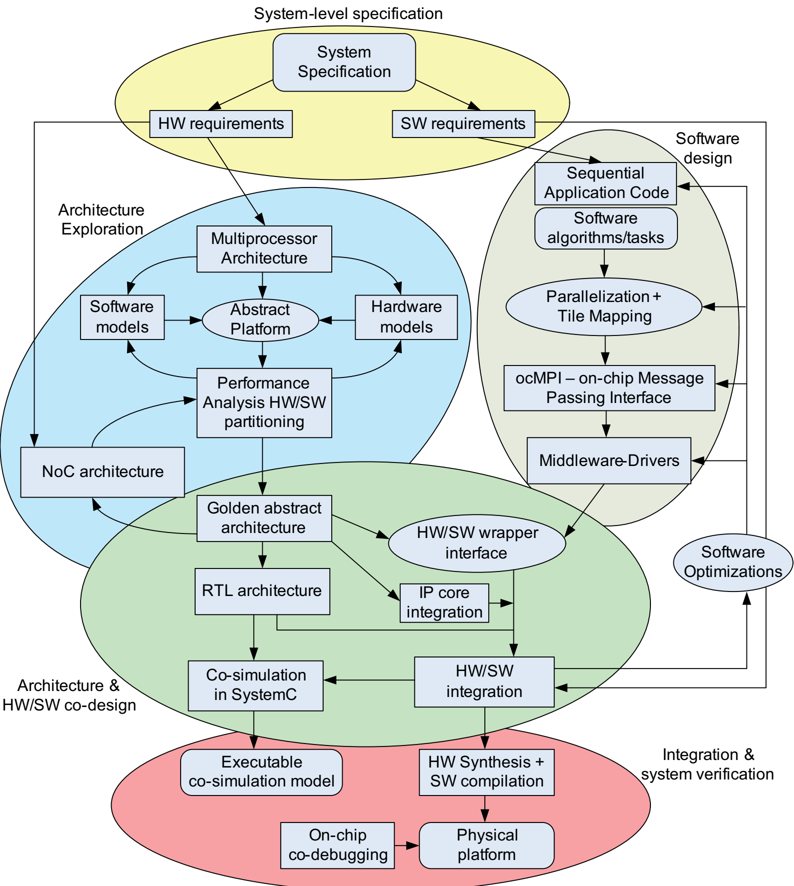


Figure 1.4: HW-SW communication-centric platform-based design (adapted from [43])

Figure 1.4 shows the HW-SW communication-centric platform-based design to implement NoC-based MPSoCs. In this design flow we can distinguish five design phases:

1. System specification: related to system-level specs, such as system and component parameters (performance, throughput, latency, bandwidth,...). At this level we impose restrictions to be checked later in terms of correct functionality and performance. It is also defined which task and components are necessary to build the complete system. Normally, this system specification is done by using a natural

language, however UML is often used for system specification, and both hardware and software designers must be present to discuss accurately the system specs.

2. Architecture exploration: its main target is to perform a co-exploration in order to decide the better architecture according to the system requirements (named in Figure 1.4 on the facing page as *Golden Abstract Architecture*). This implies that both computational components (e.g. microprocessors, VLIWs, DSPs, FFT cores, 3D accelerators, etc.) and communication infrastructure (bus-based, NoC-based or dedicate point-to-point communication) must be optimal to fulfil the initial requirements. Therefore, during this co-exploration task, it is mandatory to do a cyclic refinement phase, improving the system behaviour in each loop. During this refinement, two architectural aspects must be improved. From the point of view of computational components, it is important to decide which components will be implemented in software and in dedicated hardware, whereas from communication point of view the most important issue is to do a design space exploration to obtain the better communication infrastructure. Afterwards, system designers define the hardware-software partitioning and application mapping in order to perform a concurrently HW-SW co-design.

3. Software design: designers must optimize according to the application, the microprocessor interfaces, software drivers, and also when required, the operating system and middleware routines. The study and task mapping of the application represented with a directed acyclic graph (DAG) must be done, as well as the potential parallelization of the sequential into a parallel program, as presented in *Multiprocessing Parallelization Assistant (MPA)* in IMEC [44]. This software parallelization implicitly in conjunction with the architecture will determine the parallel programming model. A software optimization could be done in all levels of software, for instance, by redefining concurrent tasks, improving HW-SW interfaces, using hardware accelerators to boost software execution, etc.

4. Architecture design & HW-SW co-design: the golden reference model of the abstract architecture (obtained during architectural exploration) must be implemented with the help of a HDL in order to obtain a synthesisable RTL description. Obviously, this architecture will be composed by hardware IP cores, software components (described during software design) and the communication infrastructure. In addition,

an important element that must be implemented is the HW-SW wrapper interface. As commented before, through this interface we communicate the hardware platform and the software components (i.e. drivers and middleware APIs) of our system. The design of this interface is a key challenge to obtain a good system performance. Another important issue during design is to guarantee the correct operation. Once we have all system components of our MPSoC (HW or SW computational components, NoC hardware communication architecture and software APIs), it is time to integrate everything in order to implement the complete system. Another interesting path to implement the complete system without using any final physical platform is to merge all components using SystemC™ [45,46]. This language let us to describe both hardware architecture and software components in C/C++, and after that, we are able to do a co-simulation to verify the whole virtual system at different abstraction levels.

5. Integration & system verification: during last phase of the design flow, it only remains the synthesis of the systems, and the compilation of software components and libraries together with the application (or benchmarks). Depending on the environment the hardware synthesis and the software compilation can be done with multiple tool chains. In this dissertation, we used Altera® and Xilinx® design flows, and gcc-like or Java SW development tools.

   Finally, the last phase, in most cases a laborious phase, is to perform an exhaustive full-system verification getting experimental results from our applications or benchmarks. This can be done by using SystemC™ or JHDL®co-simulation frameworks, or using on-chip co-debugging through embedded logic analyzers (such as SignalTap II from Altera®, or Chipscope from Xilinx®). Initial verification of designed hardware, IP cores and our NoC design have been done using can be also performed by using ModelSim [47,48] or GTKWave [49].

The author would like to stress that NoCs are a relatively recent hot topic field on research, with roots dating to 2001, meaning that research opportunities were (and still are) numerous.

This dissertation presents work jointly carried on by the author and by several co-authors. Further, describing the complete framework in which the author's research was performed is helpful for a much better understanding of the goals and the achievements of this effort. Appropriate credit to major co-authors will be provided across this dissertation. A list of the

papers on which this work is based, complete with the names of all the co-authors, is supplied in Appendix A on page 183.

This dissertation merely reflects the research choices done by the author and his co-authors based on the information and analyses available to them (which will be substantiated wherever possible across the present dissertation), and based on time constraints.

1.4 Dissertation Outline

In Chapter 1 on page 1, we provide a brief introductions, and analysis of the main design challenges to enable high-performance computing on NoC-based MPSoC.

In Chapter 2 on page 19, we provide the state-of-the art in NoCs. This analysis lead us to the necessity to have EDA tools because the NoC design space is huge. In this chapter we show an overview of our rapid prototyping and simulation experimental framework called xENoC [50] to design custom NoC-based systems, and our tool called NoCMaker [51] is presented in detail. NoCMaker can be configured in many aspects, including topology, amount of buffering, packet and flit width, and even flow control and arbitration policies. This chapter specially deals with NoC simulation infrastructure and performance evaluation, in terms of throughput and latency, but also its impact in the area resources and power consumption. However, our framework can be also configured with many different core interfaces to enable IP core reuse. Finally, this chapter shows the potential capabilities of NoCMaker to create highly scalable parallel systems based on NoCs. It presents the work we did to physically implement full-featured NoC-based MPSoCs on FPGAs prototyping platforms using the Nios II soft core processor.

The following two chapters describes specific extensions on ×pipes NoC library [52,53] to support key advanced features for non-homogeneous traffic on NoC architectures.

Chapter 3 on page 57, is focused on the design and extension of ×pipes Network Interfaces (NIs) to interact transparently in a heterogeneous system with many IP cores compliant with different bus-based standards (i.e. AMBA 2.0 [14] and OCP-IP [12]). It will lead to seamlessly accommodation of a variety of IP cores in the system. To understand the concepts, we provide a brief discussion of OCP-IP and AMBA 2.0 protocols, but also its implication and particularities when they are interfaced on NoC systems.

In Chapter 4 on page 85, we describe in detail the architectural changes done on switches and NIs to provide QoS support at NoC-level. Basically,

we introduce two levels of QoS: *(i)* differentiation traffic services by defining a configurable priority levels scheme to obtain soft QoS guarantees, and *(ii)* a circuit switching scheme (based on channel reservation/release) on top of a wormhole packet switching NoC to deal with hard real-time QoS traffic. According to each QoS level, low-level and middleware API software routines are provided to control the QoS present at NoC-level from the application at runtime. To make this work understandable, this chapter explains the general concepts about QoS, and the related work on NoC systems. All HW-SW components will be integrated in MPARM [54], a SystemC-based cycle accurate simulator of NoC-based MPSoC using ×pipes in order to verify and test a complete system. Results in terms of area overhead and latency are presented under synthetic and real application workloads.

In Chapter 5 on page 107, we show a full-featured HW-SW design of a Floating Point Unit (FPU) on a NoC-based MPSoC based on ARM Cortex-M1. This work presents a complete HW-SW design on how to implement floating point instructions in a transparent way from/to a decoupled memory-mapped FPU coprocessor (as a bus-based element or NoC tile) without doing changes on the ARM compiler. This chapter also analyses the HW-SW synchronization and communication schemes, as well as the possibility to share the FPU between different ARM Cortex-M1s. Quantitative results are reported in terms of area and performance, as well as speedups against software floating point libraries, in different scenarios.

In Chapter 6 on page 133, we explore how to map parallel programming models, specially based on MPI on top of NoC-based MPSoCs architectures. This chapter shows and special features at architectural level in order to obtain an efficient inter-processor communication though message passing. However, in this chapter, we also explore on how to integrate our QoS features presented in Chapter 4 on page 85, on top of an OpenMP runtime library or at application level. Profiling results and speedups are reported by executing benchmarks and application on different platforms using both parallel programming models.

CHAPTER 2

Experimental Framework for EDA/CAD of NoC-based Systems

This chapter[1] describes in detail the huge design space present at the moment of design NoCs and its implications. Next, this chapter lists the state of the art on well-know NoC-based communication centric EDA tools. Finally, our experimental environment for automatic NoC design is presented. This framework is based on two tools, NoCWizard and its evolution, NoCMaker, a cross-platform open-source EDA tool to design space exploration of NoC systems. Part of this dissertation relies on these architectural designing environments.

2.1 Networks-on-Chip: Preliminary Concepts

This section covers basic concepts of NoC architectures that have not been discourse yet. First, a component based view will be presented, introducing the basic building blocks of a general NoC-based system. Then, we describe in detail all important points of the huge NoC design space, and we shall look at system level architectural issues relevant to design parallel NoC-based MPSoC architectures. After this, a layered abstraction view will be presented according to particular network models, in particular OSI, and its adaptation for NoCs. Next, we will go into further details on specific NoC research and EDA tools in Section 2.3 on page 35, doing a review of state of the art. After, we will introduce our experimental NoC design framework

[1]The author will like to acknowledge the contributions of David Castells, Sergio Risueño and Eduard Fernandez on NoCMaker tool, as well as Oriol Font-Bach on NoCWizard on xENoC framework.

listing and describing all included features to evaluate, simulate, debug and validate multiple NoC-based MPSoCs architectures. Results will be shown by generating multiple parallel Nios II NoC-based MPSoC architectures, by applying different architectures and NoC switching modes.

2.1.1 NoC Paradigm - General Overview

Figure 2.1 on the facing page shows a general overview of a NoC-based system. Instead of busses or dedicated point-to-point links, a more flexible, segmented and scalable NoC backbone is built. Basically a NoC is a ”sea” of switches which interconnect ”tiles” along the chip using ”short” point-to-point communication links. Thus, a NoC-based system is built using only three basic components:

- Network Interface (NI): there are too many ways to call this network interface in NoC literature. It can be called as Network Adapter (NA), Resource Network Interface (RNI) or Network Interface Controller (NIC), but in the essence are the same component. NIs implements the interface between each computation or memory node and the communication infrastructure. Thus, their function is to decouple computation (i.e. processing elements tiles with IP cores) from communication (switches) by adapting the potential multiple clock domains on the system (i.e. in GALS, mesochronous, etc). This element is also in charge to adapt bus-based protocols (i.e. AMBA, AXI, OCP-IP) and generate network packets in transparent way from the point of view of the user.

- Switch nodes: also called routers, are in charge to forward data according to particular switching communication technique (see Section 2.1.4 on page 26). On the switches is also implemented the routing protocol (see Section 2.1.3 on page 24 for further details) and the buffer capabilities (see Section 2.1.5 on page 28). These parameters characterize the final NoC system, in terms of area costs, power consumption and performance (i.e. bandwidth, latency).

- Links: are dedicated point-to-point connections that provide the communication between each pair of components. This links in a physical implementation represents one or more logical channels of wires.

Another important point is that, at hardware level, the presented NoC in Figure 2.1 on the facing page can be implemented using asynchronous (GALS) [55], fully synchronous [52,53] or mesochronous [56,57] logic.

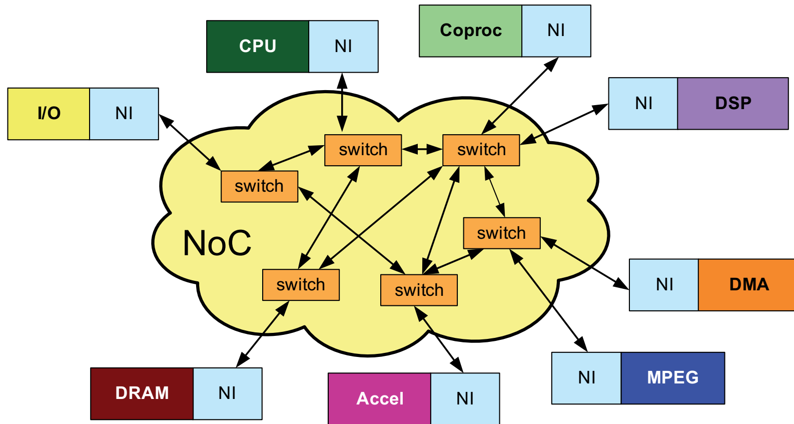


Figure 2.1: Logical view of a NoC-based system

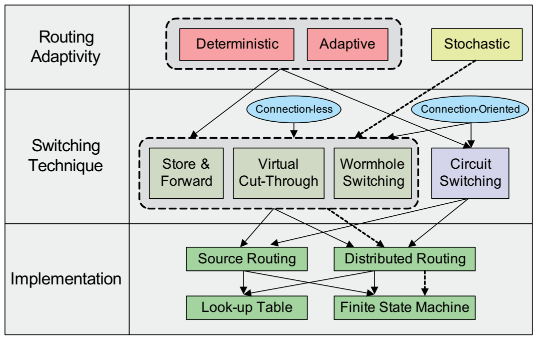


Figure 2.2: Classification of NoC design choices (from [58])

As it will show next, NoC design space is huge. Some communication NoC architectures decisions are plotted in Figure 2.2. Thus, topology, channel width, buffer sizing, floorplan/wiring, routing algorithms, switching forwarding techniques and application IP core mapping are application (and traffic) dependent are open problems that designers must select accurately when implementing a NoC architecture for a given domain.

2.1.2 Topologies

Since the emerging NoC architectures have been proposed to replace bus-based approach for large SoC designs, many on-chip network architectures have been designed adopting many topologies previously used in parallel multi-processor systems, but applying important constraints inherent to the limitation imposed in on-chip environments (e.g. limited buffer resources, power usage, flow control, routing schemes,...).

A large body of research works exist in synthesizing and generating bus-based systems [59]. While some of the design issues in the NoCs are similar to bus based systems (such as link width sizing), a large number of issues such as finding the number of required switches, sizing the switches, finding routes for packets, etc. are new in NoCs. Another important design point in the moment of design NoCs is the selection of the topology [60,61].

Hence, the topology has effect on all NoC parameters, such as, network latency, performance, area, power consumption [62]. In [63] explains the topology effects of floorplan routability of different topologies on FPGAs/ASIC. Furthermore, the topology selection is also very significant in the moment of mapping the IP cores and the tasks along the network [64]. Hence, it is appropriate that these tiles/tasks will be as much closely as possible in order to collaborate effortlessly.

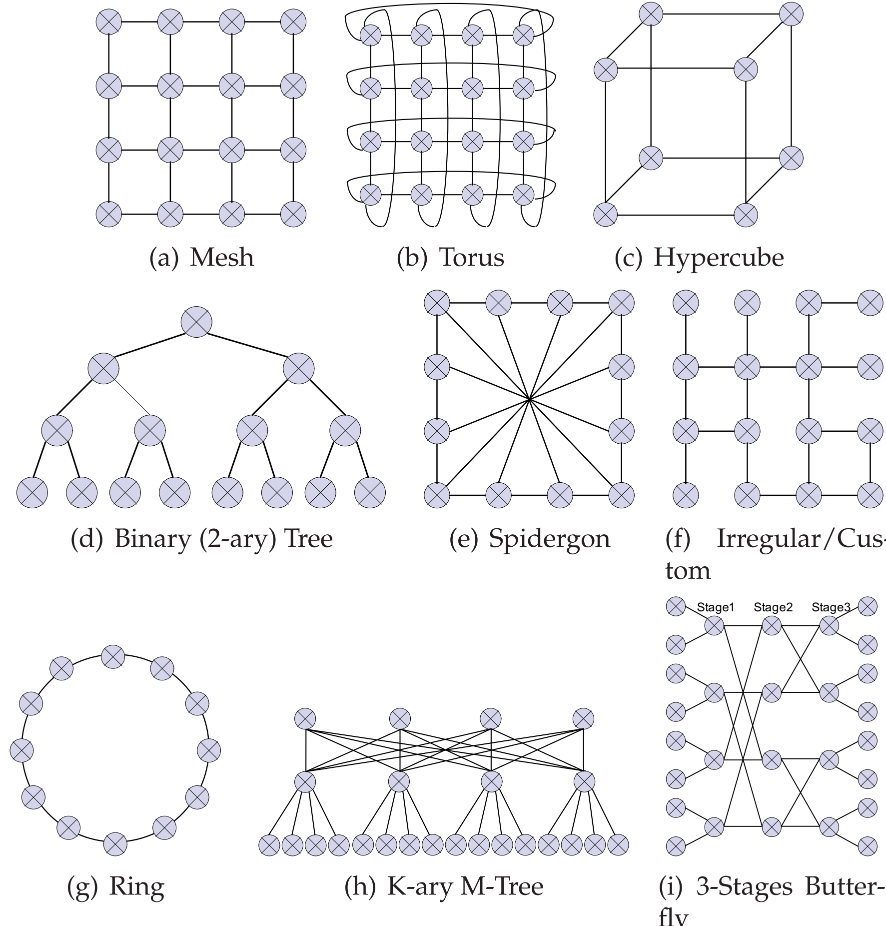
Figure 2.3 on the facing page shows a large representative subset of typical topologies used in NoCs. Generally all this subset of topologies may be classified using two different ways. These criterions are the followings:

- Regular or Irregular (or application specific).
- Direct or Indirect.

Direct topologies are those that have at least one core/tile attached to each node, whereas the *indirect topologies* are networks that have a subset of nodes not connected to any core. In that case, these nodes only perform forwarding/relaying operations (e.g. Figures 2.3(d), 2.3(h), 2.3(i) on page 23).

Early works on NoC topology design assume that using regular topologies (e.g. Figures 2.3(a), 2.3(b), 2.3(c), and 2.3(e) on page 23) like in traditional macro-networks, we can predict the performance, power consumption, area occupation and they have a predictable layout/floorplanning which helps the place and route [60,65]. However, this is only useful for homogeneous MPSoCs, but they are not suitable for application specific heterogeneous NoC-based MPSoCs, where each core has different size, functionality and communication requirements. Thus, standard topologies can have a structure that poorly matches application traffic, so the NoC system will be oversized in terms of power consumption and area costs.

In fact, until now many early existing MPSoCs (like Philips Nexperia [21], ST Nomadik [24], or TI OMAP [23]) shown in Table 1.2 on page 8 are designed statically or semi-static by mapping task to processors and hardware cores. Furthermore, to achieve more complex interconnection schemes as in traditional networks, a combination like Ring-Mesh [66], or another configuration can be used.



(a) Mesh (b) Torus (c) Hypercube (d) Binary (2-ary) Tree (e) Spidergon (f) Irregular/Custom (g) Ring (h) K-ary M-Tree (i) 3-Stages Butterfly

Figure 2.3: Representative subset of NoC topologies

Unfortunately, NoCs do not have a perfect geometry of regular grids or trees because of the inherent heterogeneity. Figure 2.4 on the following page shows a classification of those NoC systems categorized by homogeneity (X axes) and granularity (Y axes).

There is consensus that, for generic homogeneous or static multi-processing, regular topologies are suitable enough, and they can be a solution. Nevertheless, custom NoC topologies [61, 68–70], which satisfies the requirements of the emerging heterogeneous MPSoC systems should be provided as an efficient alternative solution for application-specific of NoC-based MPSoCs.

Depending on the computational nodes and its functionality, NoCs can have different granularity (as we shown in Figure 2.4 on the next page), and therefore, can be coarse or fine grain. For example, if each tile is inte-

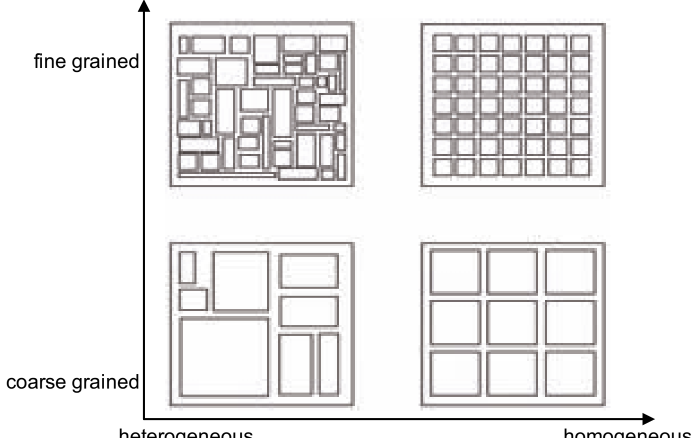


Figure 2.4: Classification of NoC systems (granularity vs. homogeneity) (from [67])

grated by a whole bus-based architecture with several CPUs, DSPs, or huge VLIWs, we can talk about coarse grain systems, whereas, when every tile is composed by relatively simple specific functions blocks, such as ALUs, FFTs, accelerators, then it became a fine grain NoC-based system.

2.1.3 Routing Protocol and Deadlock Freedom

After explaining the topology issues and NoC components, this section introduces the routing schemes supported by this topologies. The routing protocol concerns the "intelligent" strategy of moving data through the NoC using a determined switching technique (see Section 2.1.4 on page 26). The routing protocol is an important design space point because it has a strong influence in many issues, such as, the packet delivery latency under congestion, the power consumption, etc. Hence, the routing establishes a trade-off between network performance, power consumption and also area occupation. As a consequence, the selection of the right routing algorithm is crucial.

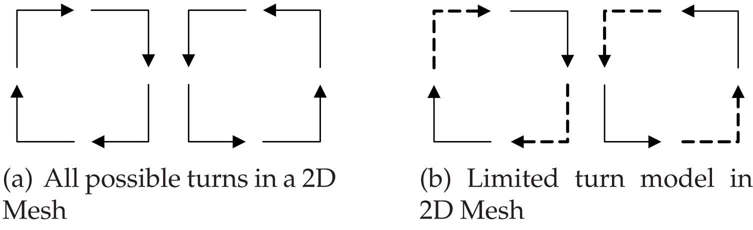
- Deterministic: in a deterministic routing strategy, the path is determined by its source and destinations address (e.g. XY-YX [71,72]).

- Adaptive or partially adaptive: the routing path is decided on a per-hop basis, and involves dynamic arbitration mechanisms (e.g. based on local link congestion or a general traffic policy). Examples of adaptive routing are Odd-even [73], West-first, North-last, Negative-first [71] or Dyad [74]).

Thus, an adaptive routing offers better benefits against deterministic routing (e.g. dynamical traffic balancing) but is more complex in its implementation. This is because using an adaptive routing may occur a deadlock or livelock. A *deadlock* is defined as a cyclic dependency among nodes requiring access to a set of resources, so that no forward progress can be made, no matter what sequence of events happens. *Livelock* refers to packets circulating the network without ever making any progress towards their destination (e.g. during non-minimal adaptive routing).

Two classes of deadlocks can occur in transactional NoCs: *routing-dependent* deadlocks and *message-dependent* deadlocks [72,75]. To avoid malfunctions both types of deadlocks must be solved properly.

Methods to avoid routing-dependent deadlocks or livelock can be applied either locally, at the nodes with support from service primitives, for instance implemented in hardware, or globally at NoC level. It is possible to implement a deadlock-free routing algorithms in Mesh topology if at least two turns are forbidden [71,76] (dotted lines in Figure 2.5(b)), as well as irregular Meshes [77,78]. This is sufficient to avoid routing-dependent deadlocks on regular topologies. However, in [68] and [79] are presented techniques to ensure deadlock-free on application-specific or custom NoC topologies. In [80,81] is reported a theory to avoid deadlocks in wormhole networks (see Section 2.1.4 on the following page), whereas in [82] a similar approach is described on a fault-tolerant scenario.



(a) All possible turns in a 2D Mesh

(b) Limited turn model in 2D Mesh

Figure 2.5: Turn model for deadlock-free adaptive routing

On the other hand, message-dependent deadlocks occur when interactions and dependencies appear between different message types at network endpoints, when they share resources in the network. Even when the underlying network is designed to be free from routing-dependent deadlocks, message-level deadlocks can block the network indefinitely. In [72,75] is presented how to solve this problem in traditional multi-processor network, whereas in [83,84] an approach for application-specific SoCs is reported. In [83,85] is described how to solve message-dependent deadlocks on SoC by quantifying its overhead in the overall system. The common technique

to avoid message-dependent deadlock is to ensure the logical (e.g. using virtual channels) or physical (i.e. using a doubled network) separation of the network for different message types when the end-to-end control mechanism is applied. This guaranteed that different message types (e.g. request messages or response messages) do not share the same resources between to endpoints.
The routing can be also categorized as:

- Minimal vs non-minimal routing: a routing algorithm is minimal if it always chooses among shortest paths toward the destination otherwise it is non-minimal. Minimal routing (e.g. XY-YX) is freedom of deadlock and livelock and deliver the packet with low latency and low energy consumption, however, are less flexible when a switch is congested. On the other side, a non-minimal routing offers great flexibility in terms of possible paths when congestion occurs but can lead to livelock situations and increase the latency to deliver the packet.

- Source vs distributed routing: in source routing the whole path is decided at source switch. In distributed routing, most common for segmented interconnection networks, the routing decisions are made locally in each routing node.

2.1.4 Switching Communication Techniques

From the point of view of hop-to-hop communication, NoCs adopt the switching techniques used in the past in traditional multiprocessor networks but applying the inherent constraints related to the on-chip scenario. The switching mode together with the routing algorithm are the main important issues to obtain certain NoC requirements. It determines a trade-off between implementation complexity (typically area costs and power consumption) and the overall network performance.
The commonly switching modes strategies used in NoCs are presented below:

- Circuit switching: involves the circuit creation from source to destination being setup and reserved until the transport of data is complete. Circuit switching can ensure a number of constant bit rate and constant delay connections between the nodes because of their exclusive use of the resources for the duration of the communication. In other words, circuit-switched NoCs [86] provides inherent QoS guarantees [87–89] and minimum buffering.

- Packet switching: forwards data to next hop by using the routing information contained in the packet. In each switch the packets are queued or buffered, resulting in variable delay depending on congestion, routing algorithm, switch arbitration, etc. Packet switching networks [13, 90] are normally used to best-effort traffic. The amount of buffering depends on the packet switching technique, and they are used to increase the performance delaying the congestion.

Other communication-based classification is related about the connection of the components. Basically there are two mechanisms, a *connection-oriented* [91] or *connection-less* [92]. The first one involves a dedicated logical connection path being established prior to data transport. The connection is then terminated upon completion of the communication. In connection-less mechanisms the communication occurs in a dynamic manner, with no prior arrangement between sender and the receiver. Thus, circuit switched communications are always connection-oriented, whereas packet switched may be either, connection-oriented or connection-less.

Another important design point in packet switched NoCs is the switching technique (see Figure 2.6 on the next page). The most common are store-and-forward, virtual cut-through and wormhole [72] and are explained below:

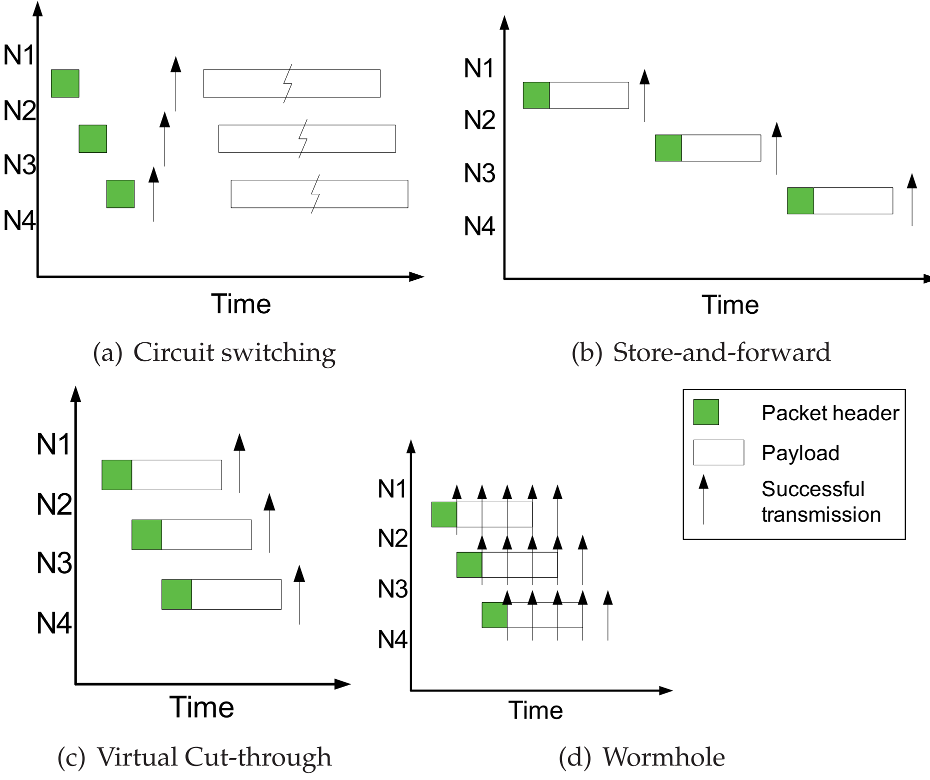
- Store-and-forward: is a packet switched protocol in which each switch stores the complete packet, and forwards it based on the information within its header. Thus the packet may stall if the switch in the forwarding path does not have sufficient buffer space.

- Virtual cut-through: is a forwarding technique similar to wormhole, but before forwarding the first flit of the packet, the switch waits for a guarantee that, the next switch in the path will accept the entire packet. Thus, if the packet stalls, it aggregates in the current switch without blocking any links.

- Wormhole: this packet switching communication scheme combines packet switching with the data streaming quality of circuit switching, to attain minimal packet latency/delay. In wormhole switching, each packet is further segmented into flits. The header flit reserves the switch channel of each switch on the path, then the body flits will follow the reserved channel like a ”worm”, and finally, the tail flit release the reservation of the channel. One major advantage of wormhole routing is that it does not require the complete packet to be stored in the switch while it is waiting for the header flit to route

to the next stages. Wormhole switching not only reduces the store-and-forward delay at each switch, but it also requires much less buffer space. The drawback is that one packet may occupy several intermediate switches at the same time.

In consequence, the amount of buffering resources in the network depends on the implemented switching technique. In Table 2.1 on the facing page is presented a summary of the latency penalty and buffering cost in each switch for every packet switching techniques.



(a) Circuit switching (b) Store-and-forward

(c) Virtual Cut-through (d) Wormhole

Figure 2.6: Packet transmission using different switching techniques (from [93])

2.1.5 Buffering and Virtual Channels

As we can see in Table 2.1 on the next page, buffers are necessary on the switches to store flits or packets specially in packet switched NoCs. Buffers contribute on the main part of area usage of the switch, therefore, it is to try to reduce as much as possible the amount of buffering area without losing performance requirements [94] at the moment of design the switches.

Protocol	Per switch cost		Stalling
	Latency	Buffering	
Store-and-forward	packet	packet	At two nodes between the link
Virtual Cut-through	header	header	At the local node
Wormhole	header	flit	At all nodes spanned by the packet

Table 2.1: Buffering costs (in terms of buffering, latency, and stalling) for different packet switching protocols (from [67])

There are two main aspects related to the buffering capabilities:

- Their size: the width and depth defines how many words are able to allocate on the switch.

- Their location within the switch: in the argument of input vs. output buffers, for equal performance the queue length in a system with output port buffering is always found to be shorter than the queue length in an equivalent system with input port buffering [88]. This is so, since in a routing node with input buffers; a packet is blocked if it is queued behind a packet whose output port is busy (head-of-the-lineblocking).

Another important issue that is necessary to remark is that, increasing the buffer size is not a solution towards avoiding congestion [95]. At best, it delays the beginning of congestion, since the throughput is not increased. Moreover, if the increment of performance is marginal in relation to the power and area overhead, then the selected buffer scheme is useless.

On the other hand, buffers are useful to absorb bursty traffic, thus levelling the bursts. In [96] is proposed a customization of buffer space allocation for an application-specific design, nevertheless a general optimized approach is much more flexible, and can fit more applications. In addition, combined with [96], in [97] it is described how important are traffic characteristics generated by the application in order to build a better space allocation. In [97] is concluded that, depending the physical platform resources, it is important to choose registers or dedicated memory to implement the queuing buffers.

In [88] input queuing and output queuing are combined for achieving virtual output queuing. This solution is moderate in costs (N^2 buffer, where N is the number of inputs) and power efficiency, but achieves high performance from output queuing. The decision to select traditional input queuing or virtual output queuing depends on system-level aspects like topology, network utilization and global wiring cost. In addition, another significant issue is to choose a centralized buffer pool shared between multiple

input and output ports or distributed dedicated FIFOs. Generally, we assume that centralized buffer implementations are found to be expensive in area due to overhead in control implementation, and become bottlenecks during periods of congestion.

Another way to improve NoCs, combined with buffering in packet switching forwarding techniques is to multiplex a physical channel by means of virtual channels.

Virtual channels reduce latency (improving the throughput), avoid deadlocks, optimize wiring utilization (reducing leakage power and congestion), increase network performance (minimizing the frequency of stalls), and moreover, the insertion of virtual channels also enables to implement policies for allocating the physical channel bandwidth, which enables to support QoS in applications (e.g. differentiated services).

In [98, 99] a virtual channel switch architecture for NoC is presented, which optimizes routing latency by hiding control overheads, in a single cycle implementation. In contrast, the use of virtual channels increases dramatically the area and power consumption in each switch.

2.1.6 Flow Control Protocols

Finally, it is also important to implement an efficient synchronization/flow control scheme between each network resource component, i.e. switches and network interfaces. Moreover, the synchronization must follow a compatible GALS scheme in order to create isolated synchronous regions, i.e. tiles, and global asynchronous communication.

- Credit-based: using a credit based technique, the network components keep counters for the available buffer space. When receiving a flit, the counter is decremented, and when a flit is transmitted the counter is incremented. If the counter reaches zero, the buffer is full and the switch accepts no further flits until buffer space becomes available again. A simply way to do this without counters is to check the status signals (empty or full) from buffering resources [94, 99].

- Handshake: in the handshake flow control, the upstream switch sends flits whenever they become available. If the downstream switch has available buffer space, it accepts the flit and sends an acknowledgment (ACK) to the upstream switch. Otherwise, the downstream switch has two options, drops/stall the flits and send a negative acknowledgement (NACK) delaying the flit until its retransmission [55, 99, 100].

There are two types of handshake:

- 2-phase: it is simple, efficient and high-throughput, but ambiguous.
- 4-phase: it is more complex to implement but is not ambiguous.

Despite the use of this flow control or synchronization schemes between network components, it is possible that congestion occurs due to the lack of infinity buffering capabilities, or congestion in circuit switching path.

In order to solve this unwanted effect, we can perform two mechanisms, delay or loss the packet. In delay model, packets are never dropped, this means that the worst that can happen is the data being delayed. On the other side, in loss model, when a congestion occurs we can drop the packet, and after retransmit it. The loss model introduces an overhead in the state of the transmission, successful or failed, and must somehow be communicated back to the source. Despite this fact, there are some pros involved in dropping packets, such as its resolving the contention in the network.

According to Dally [72], the hand-shake flow control is less efficient than the credit based flow control, because flits remain stored in the buffers for an additional time, waiting for an acknowledgment. However, if we use a circuit switching forwarding technique, both flow control schemes are efficient due to the inexistent buffering capabilities of this switching technique. In addition, as it is reported in [100], a handshake protocol is simpler and cheaper than a general credit-based flow control scheme, and can be implemented effortlessly on a GALS scheme. However, a particular case of credit-based, called stall-and-go, is cheaper than but the main drawback is that does not offer fault tolerance.

2.1.7 Micro-network Protocol Stack for NoCs

Next, a layered abstraction view of NoC systems is presented, looking at the network abstraction models. In particular, in [39, 101–104] the use of the micro-network stack is proposed for NoC-based systems based on the well-know ISO/OSI model [105]. This standardisation allows to abstract the electrical, logic, and functional properties of the interconnection scheme, up to system level application. The network is the abstraction of the communication among components. It must satisfy QoS requirements (e.g. reliability, performance, latency, and energy bounds) under the limitation of intrinsically unreliable signal transmission and significant communication delays on wires, and therefore it is important to define a robust NoC stack. Moreover, on NoC paradigm, the layers are generally more closely bound

than in a macro network. These issues often arise concerning a physically related flavour, even at the higher abstraction levels. Furthermore, NoCs differ from wide area networks in their local proximity and because they exhibit less non-determinism.

As shown in Figure 2.7 on the facing page the micro-network stack is divided in five generic layers which are explained below:

- The *physical layer* determines the number and length of wires connecting resources and switches. Ensures reliable (lossless, fault-free) communication over a data link. It considers crosstalk and fault-tolerance (both transient soft errors and hard errors). This layer is technology dependent and is implemented fully in hardware.

- The *data-link layer* defines the protocol to transmit a flit between the network components, i.e. between resource to switch and switch to switch). Data link layer is technology dependent. Thus, for each new technology new generation these two layers are defined.

- The *network layer* defines how a packet is delivered over the network from an arbitrary sender to an arbitrary receiver directed by the receiver's network address. More specifically, routing, congestion control, and scheduling over multiple links. This layer can also include network management, such as accounting, monitoring, and so on. It is technology dependent and designed in hardware.

- The *transport layer* is partially technology independent. The transport layer message size can be variable. The NI interface has to pack transport layer messages into network layer packets. This abstraction layer hides the topology of the network, and the implementation of the links that make up the network from the IP core protocol. This layer should provide end-to-end services over connections, through packetisation/unpacketisation, (de)multiplexing multiple transport connections over the network, in-order delivery, and so on. This services must offer uncorrupted and lossless transmission, and at the same time guarantee a certain throughput, jitter, and latency.

- The *system and application layer* is technology independent. At this level most of the network implementation details must be transparent to the end user. The application layer is in charge to host the application and SW APIs or run-time libraries which interact to the HW platform (e.g. middleware routines, OS, application concurrency and parallel computing APIs, high-level parallel programming models). Depending on

the implementation within this application layer is possible to define the session and presentation layers in order to be fully compatible with OSI. The *session layer* can be defined to combine multiple connections into high-level services, such as multicast, half duplex or full-duplex connections, but also to achieve a synchronization of multiple connections or the management for mutual exclusion and atomic transactions (e.g. test and set, swap). A common session protocol is synchronous messaging, which requires that the sending and receiving components rendezvous as the message is passed. The state maintained by the protocol is a semaphore that indicates when both the sender and the receiver have entered the rendezvous. Many embedded system applications utilize this sort of functionality to synchronize system components that are running in parallel. On the other hand, the *presentation layer* adapts the application data to the transport layer and vice versa (e.g. compression, encryption, endianness conversion, etc).

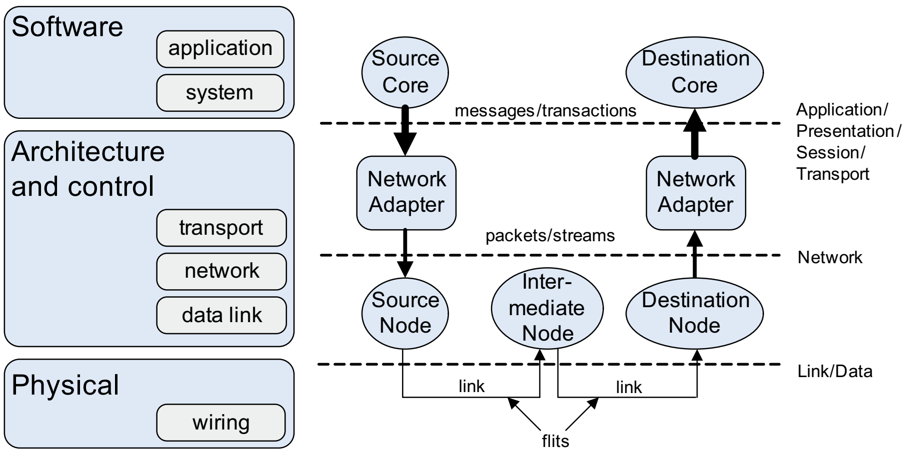


Figure 2.7: Micro-network stack for NoC-based system (from [39,67])

Using the presented abstract layered micro-network stack let us to provide a layered view of the whole systems grouping hardware and software components in a vertical approach. Therefore, thanks to this layered view is easy to provide high-level end-to-end services over connections, such as QoS requirements (see Chapter 4 on page 85), and also to achieve traffic heterogeneity in a transparent way from the point of view of the software execution (see Chapter 3 on page 57). Moreover, using this protocol stack for NoC-based systems enable the development of many different architectures, for instance, a virtual shared memory, or message passing architecture. These architectures together with its associated programming model

and APIs libraries will help SW developers to program concurrent or parallel data and computation intensive applications (see Chapter 6 on page 133). This layered approach separate optimizations of transactions from the physical layers. Thus, it is possible to simulate complex applications on a very high abstraction level, which only captures traffic events among an arbitrary number of components.

2.2 Motivation and Key Challenges

The design of future homogeneous and specially heterogeneous MPSoCs [6, 7,26,106,107] introduces many difficult problems, being the design of the interconnection, one of the most challenging ones. As the complexity of these systems grows, the design efforts are less focused on the computation and increasingly on the communication side. The outcome is a shifting towards a communication-centric designs which nowadays are focus on NoCs.

Because the design space existing on NoC paradigm is huge and the potential applications targeted to NoC-based systems will request different solutions, it is mandatory to provide EDA tools to design effortlessly and automatically a great variety of NoC-based systems. Thanks to EDA tools, we can create NoC architectures which many arbitrarily IP cores and generic or custom optimal requirements according to the application. The critical points that EDA tools must face are:

- The integration costs of the existing IP cores of many different protocols must be easy (i.e. *plug&play* composability).
- The tool must offer results of communication and traffic patterns, identification of bottlenecks, in order to quantify the performance and to balance the workload in terms of bandwidth between critical points.
- The design must be low latency within and between components (i.e. NIs and switches).
- The implementation costs must be optimal in terms of area and power consumption.
- QoS must be delivered depending on the user requirements.
- Easy capture and tuning optimization of different NoC configuration through a Graphical User Interface (GUI).
- Easy verification, simulation of the system using an automatic generated NoC self synthetic traffic.

- Power RTL code generation (i.e. VHDL or Verilog), generating pre-synthesis early area and power estimation.
- Generate a complete framework to simulate and verify hardware components together with SW components and APIs before its synthesis.
- Generate reports of main important system metrics (i.e. latency, on-the-fly packets, send/receive packets, etc).

As a starting point to create EDA or design space exploration tools we can take previous bus-based works as reported in [108] and integration tools, such as AMBA Designer [109]. However, on the on-chip network scenario many important features must be taken into account to model, verify and analyze the communication infrastructure. As a consequence, to face all challenges presented along this chapter, next we present the state of the art, and our contribution on EDA tools to design NoC-based systems.

2.3 Related work on NoC Design and EDA Tools

Once NoC concepts and its paradigm are well-established, in the last years, some efforts of many research groups, as well as companies [110–113] derived to the development of EDA tools to specify, generate, simulate, verify and evaluate NoC-based systems. Figure 2.8 on the following page shows the most relevant academic works (a exhaustive review of the state of the art is reported in [114]) on NoCs and/or EDA tools developed and distinguished by its granularity (i.e. at what level the NoC components are described) and parametrizability (i.e how easy is to change at instantiation time system-level NoC characteristics).

Some early approaches based on general purpose network simulators to evaluate NoC-based systems have been done using OPNET [115, 116] or NS-2 [117, 118]. These frameworks are useful to obtain performance metrics, but they could not be used to estimate the area and power consumption. To get this kind of information is necessary to create and automatic EDA framework, and use the HDL descriptions of the network elements. As a consequence, recently a large body of specific NoC frameworks have been reported in the last years (e.g. SPIN [13], Chain [119], SoCBUs [120], Proteo [90], NoCGEN [121], Nostrum [101] Hermes [122] and MAIA [123], Æthereal [86], MANGO [67], ×pipes [52, 53, 124] and SUNMAP [61]. Some of these frameworks, such as NoCGEN, MAIN, Proteo, MAIA, use a set of parametric template-based tool to generate NoC architectures at RTL. In ×pipes a NoC generator called ×pipescompiler [69] is able to generate

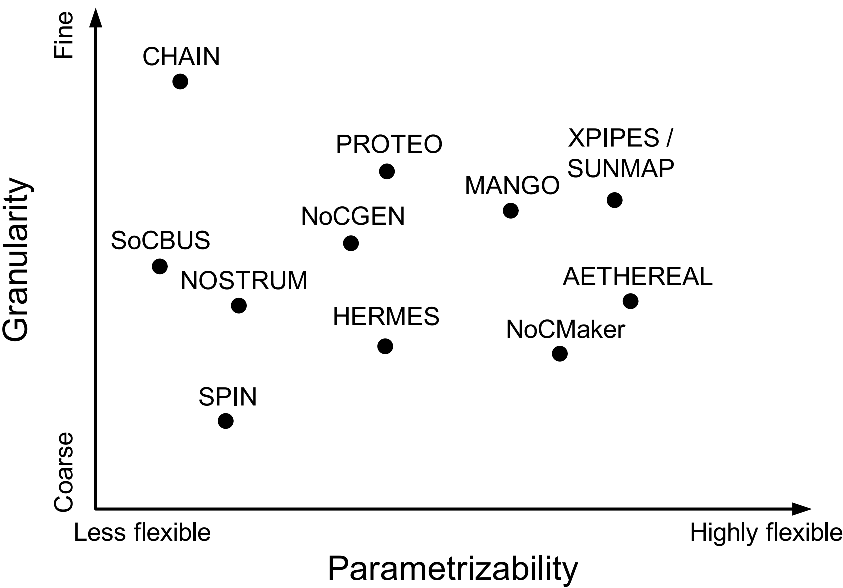


Figure 2.8: State of the art of academic EDA NoC Tools (adapted from [67])

models at Verilog or at SystemC™ RTL level as well as to simulate traffic inside the on-chip network. RTL design level is convenient for NoC synthesis and power consumption evaluation but is slow to simulate. Therefore, higher levels of abstraction models based on SystemC™ TLM can be used to achieve faster simulations times, whereas RTL is mandatory to extract area and performance metrics. Several approaches are presented in [125] for bus-based, as well as for NoCs, in [55, 126, 127].

Other initiatives focus on QoS features named as QNoC is reported in [87], whereas a completely asynchronous NoC (aNoC) is modeled using SystemC™ in [55].

2.4 xENoC environment - NoCMaker

Our contribution to the related work is our experimental Network-on-Chip environment, i.e. *xENoC* [50], to design NoC-based MPSoCs. The core of this framework is NoCMaker, a cross-platform open-source EDA tool to design space exploration of NoC-based systems. In Figure 2.9 on the next page, it is summarized a general overview of the degrees of freedom and the exploration space related to NoC paradigm. Of course, depending on the design effort and the required flexibility, we can obtain different design quality.

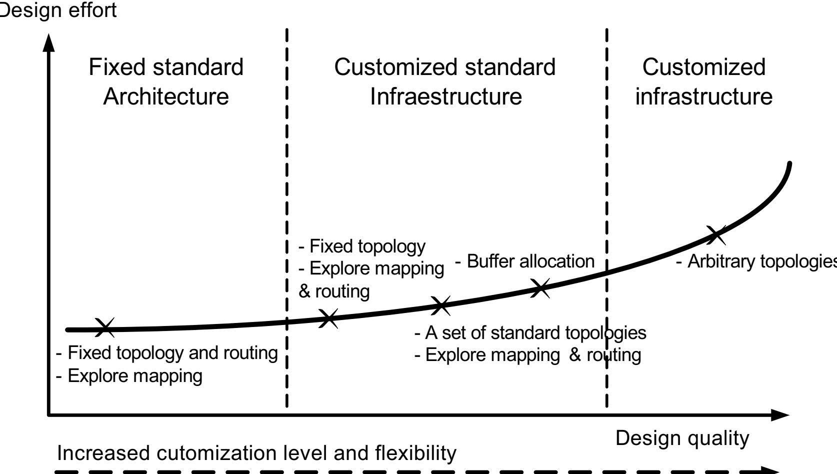


Figure 2.9: Design space exploration – design effort vs. flexibility (from [128])

Our xENoC framework and NoCMaker shares and extends some previous features shown in the related work presented in Section 2.3 on page 35. It shares the capability of ×pipes and SUNMAP to generate different topologies, and it is inspired in NoCGEN's parametrizability features to customize and select different NoC parameters (e.g. FIFO depth size, packet size, etc). In addition, NoCMaker is able to create automatically NIs as in MAIA, and it is capable to choose different tile mappings, and different routing and switching schemes (e.g. packet switching and circuit switching), special features which are not present together in any previous work.

The framework also provide the easy generation of synthetic traffic patterns, as well as the execution of real parallel applications using a parallel programming model based on MPI, i.e. j2eMPI [129] (see Section 6.5 on page 146).

2.4.1 Design Flow of NoCMaker

NoCMaker is based on Java technology, this tool let the hardware designers to create cycle accurate designs of different types of NoCs architectures. Unfortunately there are too many interrelated variables that determine a specific NoC performance: traffic distribution, network topology, switching scheme, flow control, routing algorithm, channel properties, FIFO depth, packet/flit size, etc. Thus, the NoC design space can be viewed as a multidimensional space. In some of the dimensions, only few discrete values are permitted, whereas in other dimensions infinite possible values can be

assigned.

Since NoCMaker is developed in Java, we define a Java class to represent a specific NoC design space point (NDSP). Obviously, a NDSP is a combination of different NoC parameters that identify a specific point inside all possibilities of the huge design space. As shown in Listing 2.1, the input of this class is a XML file that describes the NDSP. Instead of defining a DTD and create a conformant parser, we use an XML serialization process for much simpler solution in order to get the NDSP data. NoCMaker uses this information to automatically build NoC architectures according the selected NDSP.

Listing 2.1: XML example of a NDSP in NoCMaker

```
<org.cephis.nocmaker.model.NoCDesignSpacePoint>
  <outputChannelArbitration>FIXED_CLOCKWISE</outputChannelArbitration>
  <topology>MESH</topology>
  <queueingType>INPUT</queueingType>
  <removeInputLocalQueue>false</removeInputLocalQueue>
  <removeOutputLocalQueue>false</removeOutputLocalQueue>
  <queueLength>2</queueLength>
  <switchingMode>WORMHOLE_SWITCHING</switchingMode>
  <routingDecision>DISTRIBUTED</routingDecision>
  <routingAlgorithm>XY</routingAlgorithm>
  <flowControlMethod>FOUR_PHASE_HANDSHAKE</flowControlMethod>
  <clocking>GLOBAL_CLOCK</clocking>
  <trafficPattern>J2EMPI_PICOMPUTING</trafficPattern>
  <busType>AVALON</busType>
  <processorType>NIOS2</processorType>
  <packetSourceAddressBits>8</packetSourceAddressBits>
  <packetDestinationAddressBits>8</packetDestinationAddressBits>
  <packetMinPayloadBits>48</packetMinPayloadBits>
  <packetMaxPayloadBits>304</packetMaxPayloadBits>
  <channelWidth>32</channelWidth>
  <injectionRatio>0.0</injectionRatio>
  <injectedBytes>0</injectedBytes>
  <dyadRouting>false</dyadRouting>
  <meshWidth>4</meshWidth>
  <meshHeight>4</meshHeight>
  <masterNodes>0</masterNodes>
  <slaveNodes>0</slaveNodes>
  <emulationSupport>NO_EMULATION</emulationSupport>
</org.cephis.nocmaker.model.NoCDesignSpacePoint>
```

As shown in Figure 2.10 on the facing page, once NoCMaker parse the input XML specification, logic builder assembles and interconnect instances of different objects according to the selected NDSP in order to create a final global object with the NoC-based model. A complete set of NoC elements, which are in the repository, such as switches, processor models, traffic generators, different NIs, etc, are supplied to NoCMaker logic builder to generate a virtual prototype of the final NoC-based MPSoC.

Once the NoC model is created by NoCMaker, the hardware designer can estimate the area by analyzing the circuit hierarchy. Furthermore, if

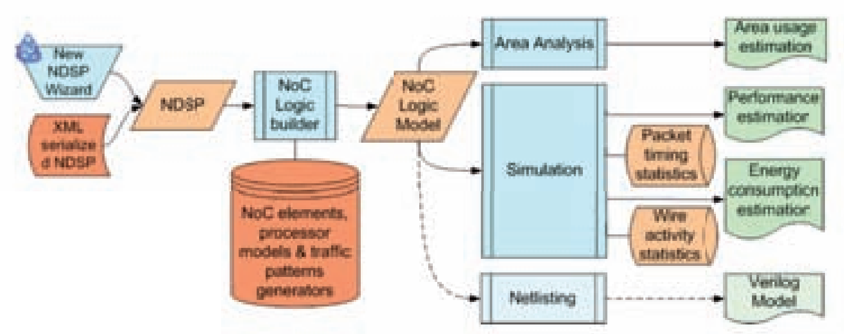


Figure 2.10: NoCMaker design flow (synthesis and metrics extraction)

a traffic pattern, benchmark or high level application is simulated, NoC-Maker will give you information about the performance (i.e. throughput and latency), as well as the dynamic power and energy consumption.

Thus, NoCMaker can be used to NoC design space exploration. During design space exploration multiple NDSPs will be created, and if the requirements are not fulfill, the above process will be repeated by changing XML specification to design a new point. Sometimes we will be able to come up with an optimal design that minimizes or maximizes our metrics, but quite often we will face trade-offs.

Finally, as the result of the design space exploration process to select the optimal architecture, NoCMaker has the possibility to generate a netlist according to the NDSP. Although JHDL only includes an EDIF netlister, we have developed a Verilog RTL netlister, so that the resulting implementation is more independent on technology (FPGA or ASIC). Several NoCs generated by NoCMaker have been synthesized as the interconnection of a MPSoC based on Nios II soft-core processor on Altera FPGAs, fulfilling the need of a correct by construction prototyping approach.

Thus, NoCMaker is able to evaluate NoC performance doing simulations, validate the system thanks to the rapid prototyping of multiple NoC-based MPSoC architectures, and finally, is able to generate Verilog RTL code that can be used to generate a real system prototype.

2.4.2 NoC modeling

The NoC modeling in NoCMaker is done either by specifying the XML presented before in Listing 2.1 or using a very intuitive step-by-step wizard GUI, which let hardware designers to choose between different options at

the moment of design the NoC-based MPSoC.

To model NoC-based systems, in this work we use JHDL [130, 131]. JHDL is a design environment that provides a Java API to describe hardware circuits at RTL in a constructive way, as well as a collection of tools and utilities for their simulation and hardware execution. JHDL provides an integrated simulation environment, which provides several facilities for exercising and viewing the state of the circuit during simulation. The simulator includes a hierarchical circuit browser with a tabular view of signals, a schematic viewer that include values of signals, a waveform viewer, a memory viewer and a command line interpreter. Moreover, the command line interpreter can be extended with custom commands that interact with NoC-based model. Thus, more complex hybrid testbenches can be developed dynamically.

There are some good reasons to use JHDL as a modeling language for NoCs:

- Circuits can dynamically change their interface using the `addPort()` and `removePort()` methods. This unique feature allows to design efficiently NoC components, specially in switches, that get rid of unused elements or ports before its synthesis.

- Block construction can be parametrize by complex arguments (like our NDSP). This feature simplifies the context information needed by the module creators and enables the design of generic traffic generators.

- Custom state viewers can be developed easily so that they provide a much richer interpretation of the NoC-based system than waveforms.

- Simulation is interactive, and can be done step-by-step at cycle accurate like in software debugging.

- It is cross-platform since uses Java Technology.

To define a design point in NoCMaker we follow the next steps:

(1) Selection of one of the topologies implemented in NoCMaker builder, i.e. Mesh, Torus, Spidergon, or a model of a partial matrix fabric similar to AMBA AHB ML [27] systems can also be selected (see Figure 2.11(a) on the next page).

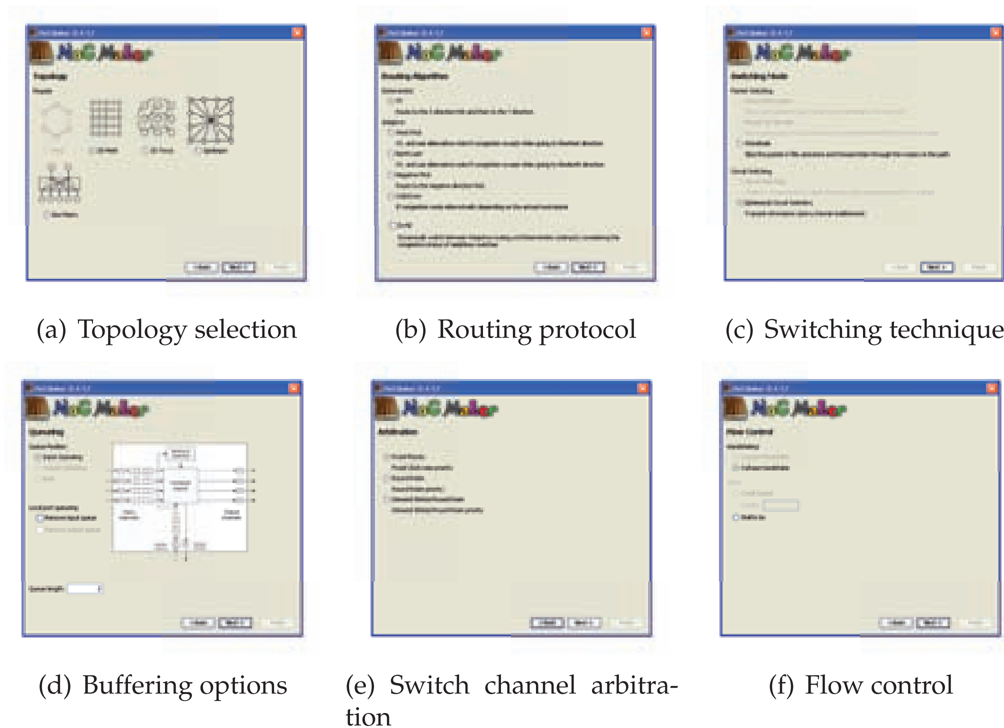
(2) Assign a deterministic or adaptive routing protocol (see Figure 2.11(b) on the facing page) under Mesh or Torus topologies.

(3) Selection of circuit switching[2], or wormhole packet switching (see Figure 2.11(c)).

(4) As long as buffers are required, depending on the switching protocol, the size and the position of them can be configured (see Figure 2.11(d)).

(5) As shown in Figure 2.11(e), the channel arbitration can be selected as fixed priority, round-robin and weighted round-robin.

(6) Selection of the NoC flow control, mainly based on *stall&go* or *4-phase handshake* (see Figure 2.11(f)).



(a) Topology selection (b) Routing protocol (c) Switching technique

(d) Buffering options (e) Switch channel arbitration (f) Flow control

Figure 2.11: Definition of a NoC design point in NoCMaker

Once the NoC backbone is modeled, several abstract processors or traffic generators can be included, as well as simple NIs models compatible with multiple standard protocols, such as FSL [132], AMBA AHB [14], Avalon [19],...) to create a full-featured simulation NoC-based MPSoC.

[2]We develop a variant of the traditional circuit switching called Ephemeral Circuit Switching [93], which is a bufferless extremely low cost scheme as is shown in Section 2.5 on page 52

2.4.3 Metrics Extraction according to Design Space Points

In order to compare different NoC designs, we have to extract some metrics from the simulated NDSPs tested by using the flow diagram shown in Figure 2.10 on page 39. To get accuracy, these metrics are extracted from the NoC JHDL RTL model.

In NoC-based systems there are some values that must be extracted to identify the efficiency of the final design:

- Overall system performance (mainly throughput and latency).
- Area usage (LUTs or gates).
- Power (mW) and energy consumption (mJ).

In NoC-based MPSoCs architectures are required high-performance, low latency, energy efficiency and low area overhead. In NoCMaker, we try to give an estimation of these metrics, but we are not wiling to create an extremely accurate and efficient models to compute them. In fact, in design space exploration is not crucial that the estimations match exactly the observed real ones, the main important issue is that the proportionality is maintained. The target is just to give an overall approximation to be able to distinct and evaluate NoC systems trade-offs.

Performance Metrics

At NoC level the performance mainly is calculated in terms of throughput and latency. To compute this metrics we follow the definitions presented in [133, 134].

The *throughput* is the maximum amount of information delivered per time unit. It can also be defined as the maximum traffic accepted by the network, where traffic, or accepted traffic is the amount of information delivered per time unit (bits/sec). Throughput could be measured in messages per second or messages per clock cycle, depending on whether absolute or relative timing is used.

In NoCMaker, the Total Throughput (TT) is computed as shown in Equation 2.1:

$$TT = \frac{Transmitted_bits}{TotalTime \times Num_{cpus}} \tag{2.1}$$

where $Transmitted_bits$ can be the packet width, or if the packet is subdivided in many flits, the flit width multiplied by the messages completed, and Num_{cpus} the number of processor on the NoC-based system.

Latency is the time elapsed (see Equation 2.2) from when the message transmission is initiated (T_{TX}) until the message is received (T_{RX}) at the destination node. This general definition can be interpreted in many different ways. If only the NoC is considered, latency is defined as time elapsed from when the message header is injected into the network at the source node until the last unit of information is received at the destination node. However, when a software message layer is also being considered, latency is defined as shown in Equation 2.3, as the time elapsed from when the system call to send a message is initiated at the source node until the system call to receive that message returns control to the user program at the destination node.

$$L_i = T_{RX} - T_{TX} \tag{2.2}$$

$$L_{sw} = sender_overhead + transport_latency + receiver_overhead \tag{2.3}$$

The latency of individual messages often is not important, especially when the study is performed using synthetic workloads. In most cases, the designer is interested in the average value of the latency (see Equation 2.4). Often, the during the execution of parallel programs, the peak latency indicates that some messages are blocked for long time, and it can be useful to track it.

$$L_{avg} = \frac{\sum_{i=1}^{P} L_i}{P} \tag{2.4}$$

In NoCMaker, we are computing the total throughput and the mean NoC latency in the network using Equation 2.1 and 2.4. These values can be computed by maintaining statistics of all the packets that go through the NoC (called on-fly packets). Thus, when an initiator processor creates a packet, it register its time stamp, and therefore, when the receiver processor unpack the packet its latency can be easily computed. In NoCMaker, these time values are obtained form the global time, which in JHDL is easily retrieved by using the method `getTotalClockCount()`.

In terms of latency, NoCMaker offers statistics related to the overall network or between point-to-point components. In Figure 2.12 on the next page is shown a complete set of statistics between different sources and destination nodes on the NoC system, displaying the mean latency, and mean bandwidth, as well as the packets transmitted.

On the other side, within the interactive simulation window (see Figure 2.13 on the following page), NoCMaker shows the overall mean latency and the total throughput, as well as the total task time (last reception).

Point to point traffic analysis

Point to point traffic characteristics

Source	Target	Packets	Mean Packet Size	Mean Latency	Mean Bandwidth
1,0	2,0	29	4	215,931	0,136
1,3	3,2	20	4	254,9	0,122
1,1	2,2	18	4	271,889	0,093
2,2	1,1	19	4	263,105	0,12
2,0	0,1	20	4	258,7	0,114
2,3	1,3	29	4	214,241	0,134
0,2	1,0	18	4	269,111	0,098
3,2	3,1	18	4	272,333	0,124
2,1	0,3	17	4	263,824	0,114
3,0	2,1	24	4	241,458	0,125
0,3	1,2	24	4	234,167	0,128
1,2	3,0	17	4	276,765	0,095
0,1	0,2	17	4	281,118	0,092
3,1	2,3	19	4	268,368	0,114

Figure 2.12: Point-to-point statistics and traffic analysis in NoCMaker

Total throughput and mean latency give a global view of the NoC performance, but it is also interesting to have detailed information of each transmission and reception at link level. This kind of information help us to understand the load in different parts of the NoC enabling the potential detection of hotspots and bottlenecks. A link load calculator is attached to each NoC link so that the traffic traversing the link can be presented graphically as shown in Figure 2.13.

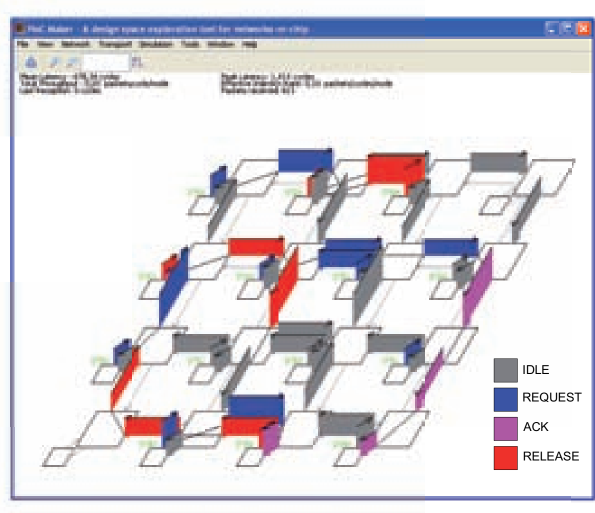


Figure 2.13: Interactive simulation window in NoCMaker

This approach has been previously applied in [135], but NoCMaker shows this information during interactive simulations, which helps a bet-

ter understanding of traffic dynamism on the NoC-based system.

In addition, Figure 2.13 on the facing page shows the four phases of the handshake flow control. So, using the interactive simulation window, we can track how packets and flits traverse NIs and switches on the NoC, how the channels are established and release, and finally, we can detect potential communication issues, such as deadlocks, livelocks, at network or at application level. On the figure, the switch is drawn as a box with their respective bidirectional ports. A line inside the box connecting an input and an output is used to indicate that this route has been established. A color code is used to inform about the state of each link according to the flow control (i.e. idle, request, ack, release).

NoC Area Usage

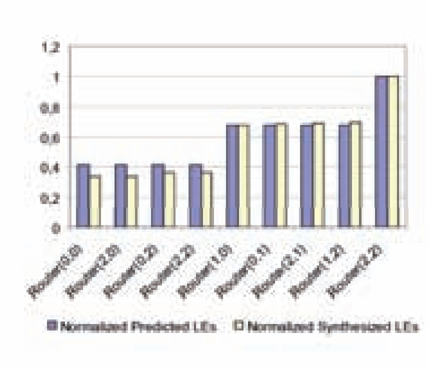
Unlike performance analysis, area usage can be obtained without doing a simulation. There are two main approaches to measure it. The first one is to use hardware synthesis, but place and route tools are quite slow, especially for very complex designs, whereas the second approach is based to synthesize some basic elements and infer the whole system area form the observed values.

Even JHDL the programmer can use TLM modeling such as in SystemC [136], JHDL encourages the use of the structural level at the moment of design a circuit but also can be use at high levels of abstraction. In NoCMaker, designers normally instantiate very simple logic elements like gates and registers to build then large blocks, which can be instantiated and reused to build even larger blocks. The result is an hierarchy logic circuit whose has a concrete number of basic elements. Area usage is usually counted in number of LUTs for FPGA, and number of gates for ASICs. Thus, if a precise profiling of each basic element is done by assigning a fixed number of LUTs or gates on each basic element, the total area can be computed by summing up this information along the hierarchy of the circuit.

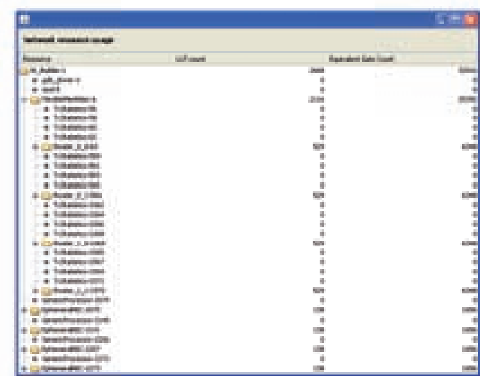
This kind of measurement is quite simplistic and should not be used to predict the real area occupancy of the final design because the synthesis tools perform resource sharing and trimming of unnecessary logic, and therefore, the area cost can overestimated. Moreover, FPGAs have different architectures of LUTs (6-input, 4-input, etc.). Nevertheless, to be useful for design space exploration, it is necessary that the early area estimations of different designs are proportional to the area usage of their real synthesized versions.

A comparison between the resource estimation and the real resource needs of the different routers on Mesh NoC using ephemeral circuit switch-

ing architectures is shown in Figure 2.14(a). The results have been normalized to eliminate the effect of the disparity between the absolute value of the estimated and synthesis results. Although resource estimations can be far from real observed resource needs, they still maintain the same proportion with a maximum worst case error of 15%.



(a) Predicted and real area usage



(b) NoC area usage window in NoCMaker

Figure 2.14: Area usage window in NoCMaker

NoCMaker includes an area usage estimation window (see Figure 2.14(b)) which shows an hierarchical report of the area costs of NoC elements, mainly NIs and routers, in terms of LUTs and equivalent gates.

Power and Energy Consumption

The calculation or estimation of energy consumption measurements is more complex of the evaluation of the performance since many factors of the underlying technology (90nm, 65nm, 40nm on ASIC or FPGAs) should be taken into account. This metric is also more complex to compute than area costs and performance metrics we basically need the whole NoC-based system at RTL (like on the estimation for the area usage), the dynamic information according the traffic patterns or application (like in the performance analysis). Thus, simulation is needed to get the activity for each wires and components of the NoC.

Several approaches have been done to calculate to forecast the energy and power consumption: *(i)* a straight approach is to use the technology provider power analysis tools using the NoC circuit and a set stimulus to compute the activity of the wires as used in PIRATE [137], *(ii)* analyze parts of the circuit and infer a cost function like in [134, 138] or *(iii)* using a predefined library like Orion [139, 140], INTACTE [141].

In NoCMaker, we consider only the dynamic power (see Equation 2.5) which is an important part of the power consumption, together with leakage and static. The dynamic power consumption can be computed by the following expression presented in Equation 2.6 derived from the classical Equation 2.5, which describes the power on a wire.

$$P_{wire} = \frac{1}{2} \cdot \alpha \cdot C_{wire} \cdot {V_{dd}}^2 \cdot f \tag{2.5}$$

$$P_{NoCdyn} = {V_{dd}}^2 \cdot \sum_{i=1}^{n} C_i \cdot \alpha_i \tag{2.6}$$

The load capacitance (C_i) is determined by the gate capacitance and drain (C_g) of all nodes connected to each node, and the wiring capacitance among them. The fan-out (F_i) is the number of connected nodes at the output node i. For intra-switch wiring we make the assumption that wire length is proportional to the fan-out, or in other words, that each connected node is attached directly to its driving node via a non-shared wire with capacitance C_w. Clock distribution makes a major contribution of the overall power and energy consumption, so clock wires have different capacitance (C_{clk}). The capacity of inter-switch wires is determined by C_{link}. In NoCMaker, C_g, C_w, C_{clk} and C_{link} are parameters of the design space. F_i is obtained by doing an analysis of the circuit structure hierarchy, whereas α_i is found during the simulation thanks to the `wire.getTransitionCount()` method. This simple approach can be use to compare different NoC or even to create a power breakdown of the system. The final formula to compute the load capacitance is described in Equation 2.7.

$$C_i = F_i \cdot (C_g + C_w) \tag{2.7}$$

When flits or packets are transmitted/received on the NoC along multiple hops, both the inter-switch wires and the logic gates in the switches and NIs toggle and this will result in energy dissipation. In NoCMaker, we are concerned in the dynamic energy dissipation caused by the message passing.

Consequently, we determine the energy dissipated by the flits in each interconnect and switch hop. The energy per flit per hop is given by the formula presented in Equation 2.8.

$$E_{hop} = E_{switch} + E_{interconnect} \tag{2.8}$$

where E_{switch} and $E_{interconnect}$ depend on the total capacitance and signal activity of the switch and each section of interconnect wire, respectively (see Equations 2.10 and 2.9). The capacitance C_{switch} and $C_{interconnect}$ have been calculated in a similar way as presented before using Equation 2.7.

$$E_{switch} = \alpha_{swtich} \cdot C_{switch} \cdot {V_{dd}}^2 \quad (2.9)$$

$$E_{interconnect} = \alpha_{interconnect} \cdot C_{interconnect} \cdot {V_{dd}}^2 \quad (2.10)$$

If we enable the profiling option to compute the energy and power consumption, NoCMaker pop-up a window with all statistics in a hierarchical view.

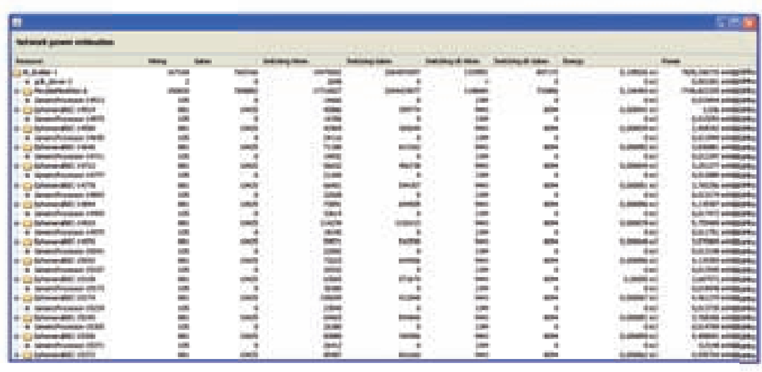


Figure 2.15: Overall power usage window statistics in NoCMaker

2.4.4 Modeling Traffic Patterns and Benchmarks

Using the XML definition, NoCMaker generates a cycle accurate model of whole system by attaching different processor models or traffic generators on each tile of the NoC.

In NoCMaker, the workload model of traffic generators is basically defined by three parameters: distribution of destinations, injection rate, and message length. As shown in Figure 2.16(a) on page 50, we can tune these parameters and defined different custom traffic patterns very easily through a GUI or importing a **.csv* stimulus file.

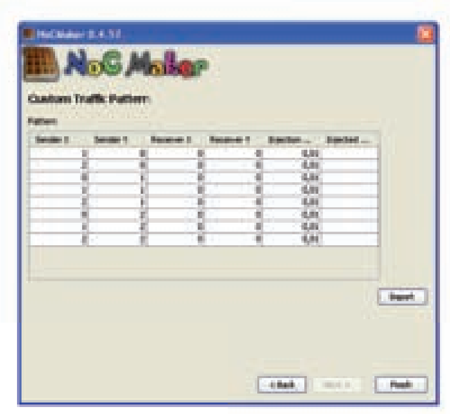
NoCMaker supports two types of executions models: *infinite* or *finite* traffic patters. In infinite traffic patters the benchmark are executed until the user can see the stability of the system. On finite, the task or the traffic pattern itself is executed until its runtime completion. NoCMaker

includes some simple but representative finite time traffic patterns, from synthetic workloads, broadcast, narrowcast, cache coherence write through token pass protocols, and synchronization ones using barriers.

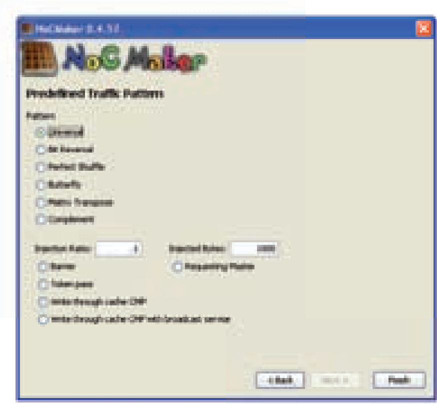
Moreover, specific temporal locality communication patterns between pairs of nodes have been implemented to evaluate the performance of the interconnection: universal, bit reversal, perfect shuffle, butterfly, matrix transpose and complement. These communication patterns take into account the permutations that are usually performed in parallel numerical algorithms. In these patterns, the destination node for the messages generated by a given node is always the same, and the utilization factor of all the network links is not uniform. However, these distributions achieve the maximum degree of temporal locality. These communication patterns extracted from [133] can be defined as follows:

- Universal: all nodes communicate with the all nodes.
- Bit reversal: The node with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ communicates with the node $a_0, a_1, ..., a_{n-2}, a_{n-1}$.
- Perfect shuffle: The node with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ communicates with the node $a_{n-2}, a_{n-3}, ..., a_0, a_{n-1}$ (rotate left 1 bit).
- Butterfly: The node with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ communicates with the node $a_0, a_{n-2}, ..., a_1, a_{n-1}$ (swap the most and least significant bits).
- Matrix transpose: The node with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ communicates with the node $a_{\frac{n}{2}-1}, ..., a_0, a_{n-1}, ..., a_{\frac{n}{2}}$.
- Complement: The node with binary coordinates $a_{n-1}, a_{n-2}, ..., a_1, a_0$ communicates with the node $\overline{a_{n-1}, a_{n-2}, ..., a_1, a_0}$.

Despite this fact, the designer can easily extend the simulator to support their own custom traffic patterns. We extend the simulator to attach different processor models or traffic generators on each NoC tile. In addition, we extend them to support parallel computing using message passing parallel programming model as we will show in Section 6.5 on page 146). In NoCMaker, it is possible to execute parallel programs using MPI [142]. Thus, we develop a j2eMPI, a Java MPI-like library to allow embedded parallel computing on NoC-based MPSoCs (further information is presented in Section 6.5 on page 146).



(a) Custom traffic patterns



(b) Pre-defined traffic patterns

Figure 2.16: Custom/pre-defined synthetic traffic patterns in NoCMaker

2.4.5 System and Circuit Validation

The building blocks to design a NoC backbone are mainly NIs and switches. These circuits can be fairly simple and easy to verify independently. However, when several modules are combined to form a full-feature NoC, the verification becomes quite complex. A simple 4x4 mesh NoC, for instance, has 80 communication links among the switches. If each link has 32-bits flit and two control bits, the number of wires that interconnect the switches is 2720. Analyzing the behavior of such a system at the inter-router level with classical waveform analysis becomes tough as, after grouping 32 data bits, you still have 240 waveforms to look at. Moreover, the network dynamics is crucial in the appearance of errors. Congestion can stall packets along different routers, or could be triggered when a very specific traffic pattern has occurred at a certain region of the NoC.
A set of techniques are proposed to tackle this high complexity.

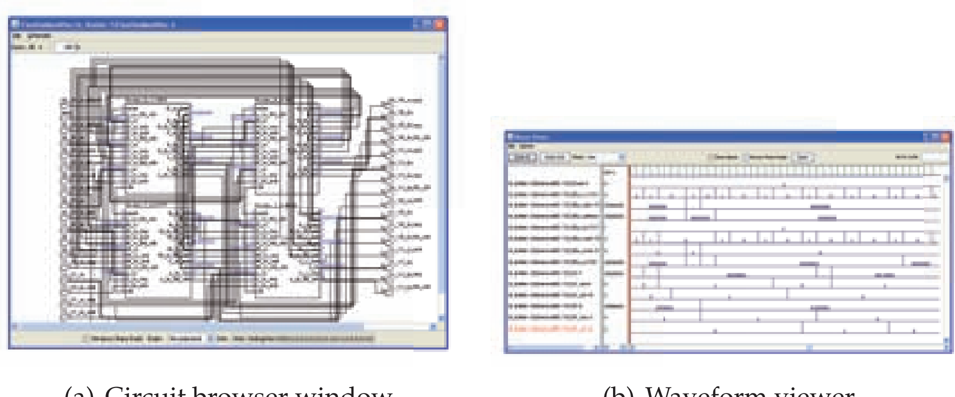
(i) Registering the time of the packet transmission and reception events in the processor models allows to compute performance information like point to point mean latency and bandwidth as showed before in Figure 2.12 on page 44, but also to present the packet communication in a timeline (see Figure 2.18 on page 52), so that the cause of big latencies can be identified.

(ii) The graphical view at TLM of the NoC topology in Figure 2.13 on page 44 can be used to give quick information of the routing decisions

and flow control status, which are two common causes of NoC malfunctions.

(iii) Detailed information from the internal circuits state can be visualized with classic JHDL facilities, the circuit browser, the interactive schematic view Figure 2.17(a) and the waveform viewer Figure 2.17(b) to verify the correctness of the simulation process. All these features presents the values of the signals of the circuit wires in real time during a simulation.

(iv) Assertions can be inserted in behavioral circuits so that the correctness of the system is constantly supervised. If some incorrect state occurs, the assertion code stops the simulation and pops up a message giving the necessary hints to the designer to identify the cause of the error.



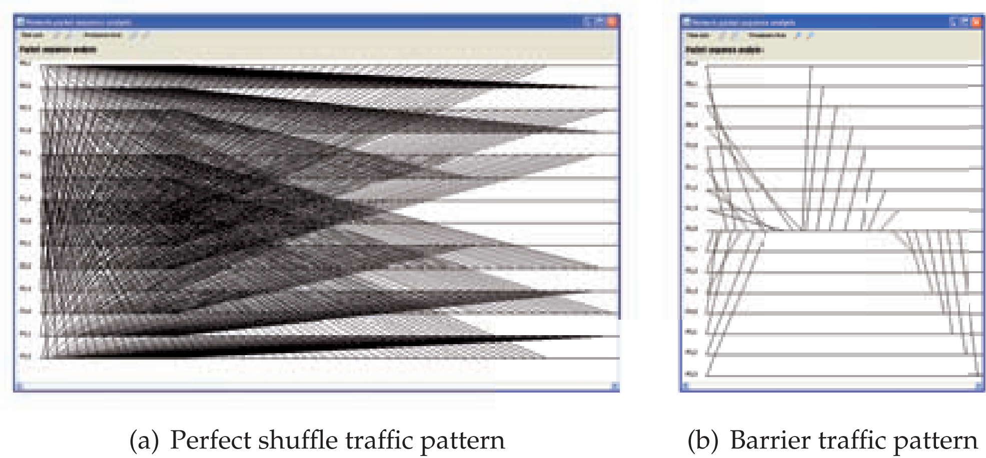
(a) Circuit browser window (b) Waveform viewer

Figure 2.17: Verification of NoC-based systems using NoCMaker

An additional feature in conjunction with the presented verification tools, is the possibility to advance one or more clocks on the simulation together with the software application using the interactive simulation windows (see Figure 2.13 on page 44). Combining all these techniques the designer can detect problems at a high level, and successively go to deeper and deeper levels of detail to identify the final error causes and correct them.

Another feature developed in NoCMaker to allow system level validation at TLM-like level is the *packet sequence analyzer*. As shown in Figure 2.18 on the following page, the packet sequence analyzer track transmitted and received messages on the NoC. This feature is specially interesting to detect application message deadlocks/livelocks when message passing applications run on top of the NoC. In Figure 2.18(a) on the next page, we show the

sequence analysis of a simulation running a perfect shuffle traffic pattern. In Figure 2.18(b), we depicted the traffic generated by `j2eMPI_Barrier`, where the root processor in the Mesh is the node with coordinates (2,0).



(a) Perfect shuffle traffic pattern (b) Barrier traffic pattern

Figure 2.18: Packet sequence analysis window in NoCMaker

2.5 Experiments using xENoC – NoCMaker

In this section we use NoCMaker to contrast a variant of circuit switching and wormhole NoC packet switching NoCs. All these synthesized NoC-based MPSoC systems are based on Nios II processor [20]. Even if NoC-Maker can be used to design space exploration, in this dissertation, the aim is only to show the rapid prototyping of NoC fabrics through NoCMaker, and how these IP core fabrics can be effortlessly integrated on Altera's design flow.

Thus in these experiments, we will integrate our NoC fabric generated by NoCMaker in conjunction with multiple Nios II on EP1S80 FPGA. As shown in Figure 2.19 on the next page our system will be composed by:

- Standard Nios II soft-core processor.
- Customize NoC fabric using NoCMaker.
- On-chip memory.
- Profile (i.e. timers, performance counters).
- Debug peripherals (i.e. UARTs/JTAGs).

From software point of view, we integrate our software stack (further information in Section 6.5.5 on page 153). Optionally, as long as it is required an embedded OS or micro-kernel, like ucLinux [143] or eCos [144], which also can be integrated on the software stack, but they are not necessary and it is out of the scope of this dissertation.

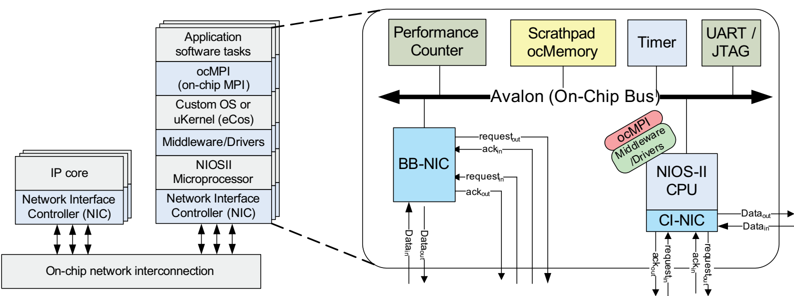


Figure 2.19: Nios II NoC-based MPSoC architecture

2.5.1 Synthesis of Nios II NoC-based MPSoC architectures

In NoCs the main forwarding techniques are circuit switching and packet switching. In this section we try to quantify the area overhead between both switching techniques. Figure 2.20 shows the differences between a pure circuit switching and our ephemeral circuit switching. Our ephemeral circuit switching is a variant of the pure circuit switching scheme, where during the set up of the channel, some data is transmitted when open channel signaling packet, and the release of the channel is done automatically, and the end of each packet.

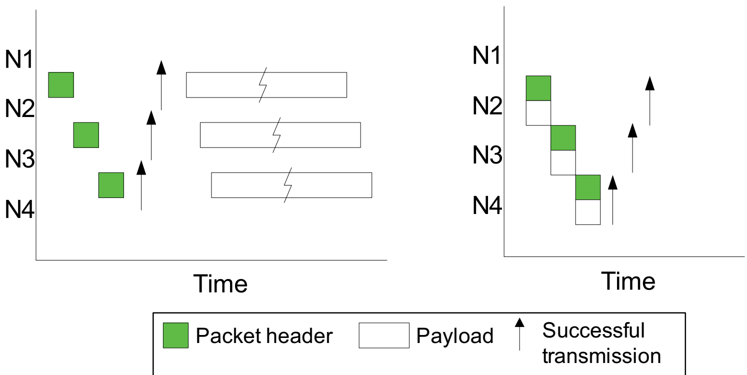
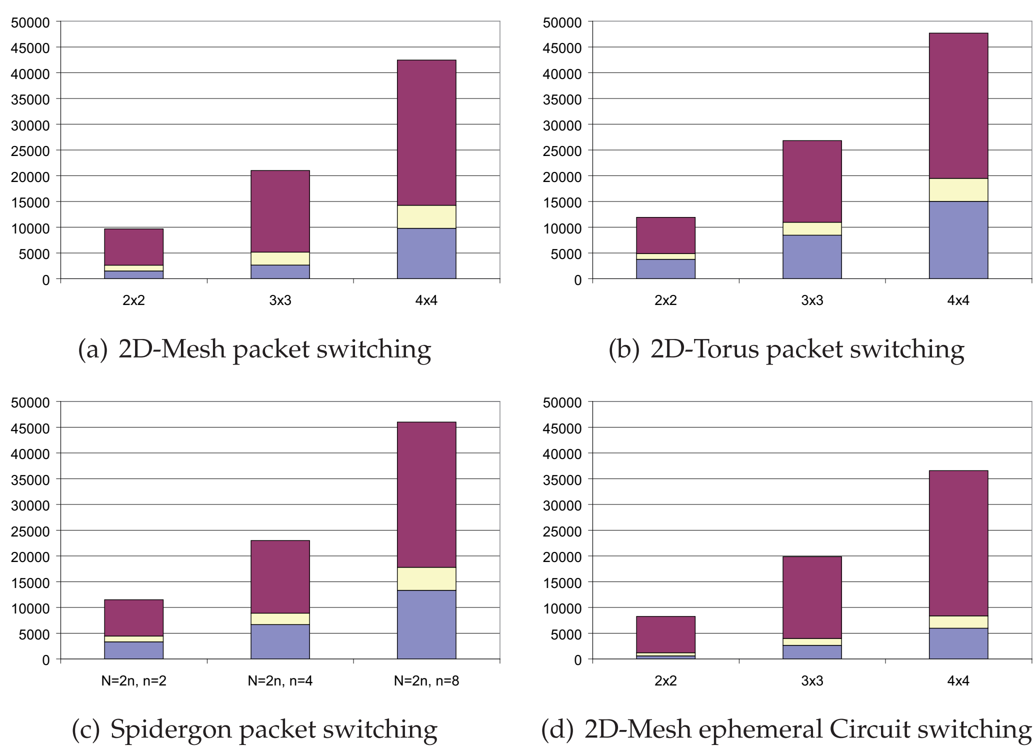


Figure 2.20: Circuit switching vs. ephemeral circuit switching

We have applied NoCMaker to generate different regular topologies: using ephemeral circuit switching [93] and wormhole packet switching. In Figure 2.21, we present the area results on a Nios II NoC-based MPSoC, using 32 bits flit width and minimal buffering (8 slots), and 4-phase handshake protocol features on packet switching NoCs.

As expected, and also predicted with the early estimation area report in NoCMaker, the 2D-Torus packet switching NoC is the most expensive in terms of area usage because every router is connected to all neighbors, whereas the 2D-Mesh ephemeral circuit switching is the cheapest.



(a) 2D-Mesh packet switching

(b) 2D-Torus packet switching

(c) Spidergon packet switching

(d) 2D-Mesh ephemeral Circuit switching

Figure 2.21: Synthesis (in LEs) of Nios II NoC-based MPSoCs

In fact, doing a fair comparison our 2D Mesh ephemeral circuit switching and packet switching in Figure 2.21(a) and Figure 2.21(d), it easy to notice that our bufferless variant of circuit switching scheme use around 15% less area than the equivalent packet switching. Spidergon, in Figure 2.21(c), is in between of both, having a similar area usage of Meshes under equivalent configurations.

Another important metric is to measure the ratio between the communication, and the computational resources. The area breakdown of a 3x3 Mesh circuit switching Nios II based MPSoC is shown in Figure 2.22. The

cores and peripherals occupies the 80% of the area, meanwhile the area costs related to the interconnection fabric (switches, and NIs) is 20%. This ratio in an equivalent 2D-Mesh packet switching (see Figure 2.21(a) on the preceding page) raise until 34% for the NoC backbone and 66% used for the computational tiles.

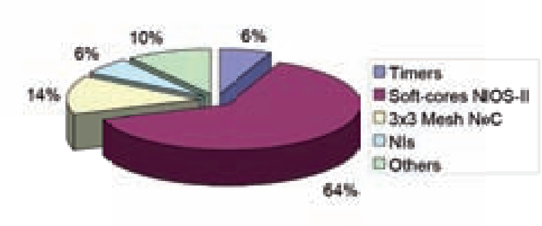


Figure 2.22: Area breakdown of our Nios II NoC-based MPSoC

Despite the low area costs of circuit switching schemes, in terms of area savings and null buffer requirements, generally this architecture is worse in terms of flit/packet contention and throughput than packet switching. Hence, the selection between one of these two switching schemes will mainly depend on high-level system and application factors. These factors come from traffic statistics, such as contention probability, and the application throughput requirements apart from the availability of physical resources.

2.6 Conclusion and Future Work

We have proposed a cross-platform rapid and robust experimental NoC-based environment to create multiple interconnection fabrics for next-generation MPSoCs. The environment is based on a repository of NoC component, enabling the re-use of communication models, which can be effortlessly extended.

NoCMaker is able to generate HDL models of the specified NoC through a simple but strong XML specification. This allows us to explore and test multiple NoC backbones, through its simulation, verification and by extracting, early estimation area costs, power consumption and the overall system performance (in terms of total throughput and mean latency).

As we shown, HDL models can be integrated effortlessly in Altera design flow [145] to create a full-featured prototype of parallel Nios II NoC-based MPSoC architecture. This is a particular case of study, because definitely the same HDL NoC model can be also included as an IP intercon-

nection fabric on AMBA Designer [109], within Xilinx design flow [146] like the work presented in [147], or Cadence SoC Encounter [148] to perform synthesis and P&R on different ASIC technologies.

Both, the xENoC environment in conjunction with NoCMaker is a stable framework that gives as the chance to do a deeper research in NoC design. In addition, xENoC and NoCMaker joins many topics and features already presented in previous related work, but it also extends the design process to include software development by adding the software components, such as j2eMPI (see Section 6.5 on page 146).

The design of NoC systems is difficult because of the huge NoC design space. Moreover, the simulation and verification of a complete NoC-based system can be very tedious because a relative small NoC can have hundreds of point-to-point wires. For all of these reasons, EDA tools, like NoCMaker are definitely necessary to effectively develop and verify NoCs at system level.

Since, embedded systems could be sometimes application specific, the next step to improve improvement of our xENoC EDA framework will be the inclusion of irregular topologies on NoCMaker. A priori, this is an easy step by means of use source routing scheme. In that case, a part of the XML, we will need a complete graph representation with the topology as an input. This will lead to enable another degree of freedom on NoC design space at the moment of perform a full process of design space exploration by generating more and more NDSPs and selecting them according input constraints. Other directions are to extend the NoC models to support virtual channels [72, 98].

CHAPTER 3

Full-Featured AMBA AHB Network Interface OCP Compatible

This chapter[1] shows the design of a full-featured AMBA 2.0 AHB Network Interface (NI) for ×pipes NoC library. This work also extends the packet format to ensure traffic compatibility and transparent plug&play in the NoC architecture of AMBA 2.0 AHB and OCP-IP cores. This development will lead to future design of heterogeneous NoC-based systems using multiple protocols. Finally, an evaluation of the AHB NI is reported, comparing the presented AMBA 2.0 AHB NI with the existing OCP NI in the ×pipes NoC library, in terms of area costs and circuit performance (i.e. f_{max}) by means of a physical synthesis on FPGA platforms.

3.1 General Concepts of ×pipes NoC library

In this section we introduce briefly ×pipes NoC library [52, 53] in order to understand the rest of the chapter. ×pipes is based on a set of architectural choices:

- A library of fully synthesizeable components which can be parameterized in many respects, such as buffer depth, data width, arbitration policies, etc. This components have been designed for lowest area, power consumption and minimum latency.
- NIs supports the OCP-IP socket standard [12]. However, in this work the functionality is extended to be compatible with AMBA 2.0 AHB standard [14] as shown in Section 3.5 on page 67.

[1]The author will like to acknowledge contributions by Federico Angiolini.

- Is fully synchronous, but it offers facilities to support multiple clock domains at NIs by including integer frequency dividers. Nevertheless, mesochronous [57] and GALS approaches are still possible by inserting synchronizers at appropriate places in the NoC.

- Routing is static and determined in the NIs (*source routing*). No adaptive routing schemes are implemented due to its complexity in terms of area, and dead-lock avoidance. Source routing minimizes the implementation cost maximizing the flexibility and its potential customization for application specific NoC systems.

- Adopts wormhole switching [72, 133] as the only method to deliver packets to their destinations. It supports both, input and/or output buffering [72, 133] according stall-and-go or handshake flow control protocols.

- QoS is provided at design time according to specific application traffic requirements defined by the application communication graph (e.g. point-to-point bandwidth) and the initial simulation and estimations. Runtime QoS features will be added and reported in Chapter 4 on page 85 on this dissertation, as a complement to design time QoS approach.

- Does not include explicit support for virtual channels [72,133], instead parallel links can be deployed between each pair of switches to fully resolve bandwidth issues.

3.2 Motivation and Challenges

The Virtual Socket Interface (VSI) Alliance [11] had developed at the end on 90s a standard interface to connect virtual components (VCs) to on-chip buses. Nowadays, a wide range of IP cores and bus-based systems and protocols (such as AMBA/AXI [14,32], OCP-IP [12], STBus [28], IBM Coreconnect [17], Avalon [19], etc) adopt or extends this standard to interconnect complex SoCs and MPSoCs.

Nevertheless, since the NoC paradigm has been demonstrated as a feasible and scalable solution to implement the interconnection fabric, on highly composable systems, a generic NoC socket interface must be designed to integrate different subsystems and protocols. Nowadays, interconnects use a multi-master and multi-slave protocols including shared or hierarchical

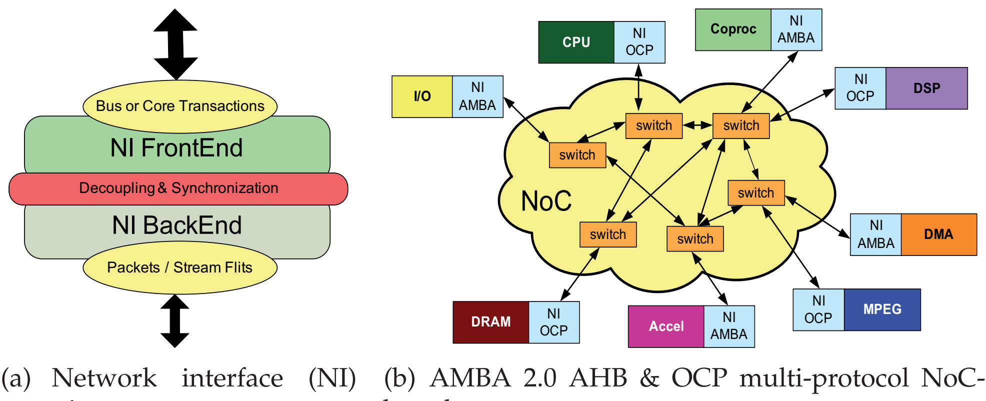
or bridged and clustered buses of several types, partial and full crossbars, etc. To make NoC-based systems meaningful and attractive, the communication between this communication resources need to be transparent, and the interface between a resource and an IP core need to be standardized. Nowadays, this is key challenge on NoC design, as well as to provide efficient and reliable communication services required from the application point of view.

A common VSIA-like or VC-like for NoCs must be standardized to enhance design-productivity by plug together off-the-shelf IP cores from different vendors and protocols in order to design custom heterogeneous multi-protocol NoC-based MPSoCs. Moreover, it will lead to enable the orthogonalization of computation and communication elements [149] to fast design space exploration, and hierarchically verification.

As a consequence, a key challenge nowadays is how to design efficient NETWORK INTERFACES (NIs) or custom wrappers which covert automatically foreign IP protocols to the NoC specific protocol (see Figure 3.1(a) on the next page), but without adding too much overhead in terms of area, and latency penalty. However, the overhead costs to include a NI are often considered negligible because it boost the reusability and productivity. When NoC packets or stream of flits are converted to core transactions the NI must be intelligent enough, identifying and adapting each protocol or type of transaction whether is required.

Similar to a computer network with layered communication protocols, the NoC relies to its own layered micro-network stack. This layered approach based on ISO-OSI protocol stack (see Figure 2.7 on page 33) allow the mixture of transactions of multiple protocols in a NoC-based system. Thus, the lowest four layers (i.e. transport layer, network layer, link layer and physical layer) are implemented in the *NI backend* (see Figure 3.1(a) on the next page) and are part of the NoC backbone and its specific protocol. The computational resources reside outside thanks to the decoupling logic and synchronization. On the other side, the *NI frontend* shown in Figure 3.1(a) on the following page is responsible to support bus or core transactions of the standard node protocol (e.g. AMBA 2.0, AXI, OCP, etc.) as well as provide a degree of parallelism, and services at session layer.

Despite this ideal view many issues and problems arise. Some of these protocols, like OCP supports non-posted writes feature not present a priori in AMBA 2.0 AHB. Others such AXI, have independent read and write channels further obscuring ordering constraints. Different semantic can also be found, for example, certain types of OCP byte enables are not supported in AMBA 2.0 AHB, while protected transactions are supported in AMBA 2.0



(a) Network interface (NI) overview

(b) AMBA 2.0 AHB & OCP multi-protocol NoC-based system

Figure 3.1: NI overview on a multi-protocol NoC-based system

AHB but does not exist at all in OCP. In addition, exclusive reads and lock transactions impact directly NoC transport level because switches must take decisions about exclusivity and lock-unlock transactions. Thus, the network and physical level is mixed with transport level and NoC services should be activated in a particular transaction packet.

Therefore, even it is not easy, NIs should control all these issues as much as possible enabling a full featured compatible transactional at session layer with many different protocols. As we shown in Figure 3.1(b), the outcome of this layered approach will enable the feasibility to create a multi-protocol NoC-based system.

Finally, the interaction of NIs with higher layers is definitely a key point to exploit NoC services, e.g. providing QoS, from the application level to the NoC backbone, as it is explained in Chapter 4 on page 85.

3.3 AMBA & OCP-IP Interconnection Fabric Overview

In this section, we focus our attention on some of the best-known, and most widely used, bus-based on-chip communication fabrics and socket architectures.

- Advanced Microcontroller Bus Architecture (AMBA) [14].
- Open Core Protocol (OCP-IP) [12].

Thanks to its moderate silicon footprint AMBA 2.0 is widely used as the fabric architecture in today's of many SoCs and MPSoCs. On the other

side, OCP-IP success because of its flexibility and due to its architectural flexibility and advanced features enabling independently IP core creation from the communication architecture. Mainly, both fabrics architectures are used to boost productivity, *plug&play* IP integration enabling IP reuse, but nowadays the challenge is to integrate them within a full-featured NoC-based system.

This section does not intended to explain exhaustive both fabrics, rather than to do a brief overview on both architectures. It will help the reader to understand the rest of the chapter, but also to show the trend of on-chip fabrics, reporting incompatibilities and specific features.

3.3.1 AHB/AXI Architecture Overview

The *Advanced Microcontroller Bus Architecture (AMBA) 2.0* [14] interconnect distinct three types of buses:

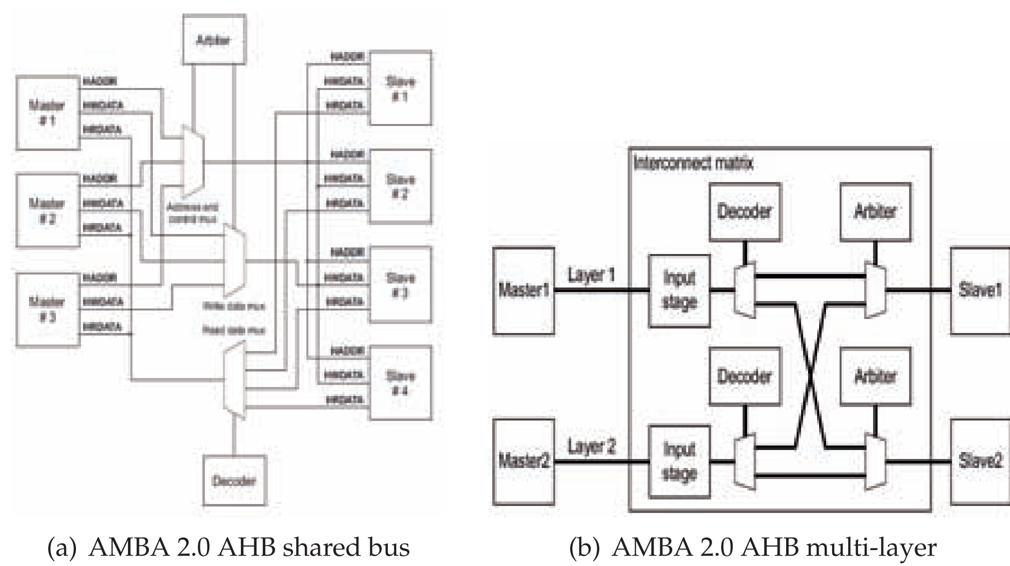
- Advanced High-performance Bus (AHB).
- the Advanced System Bus (ASB).
- the Advanced Peripheral Bus (APB).

Despite this fact, AMBA 2.0 is considered an hierarchical bus based on two levels, AHB bus acts as a high-performance backbone whereas APB is low-power low-performance bus. Thus, normally AHB includes high performance cores, such as CPUs, on-chip and off-chip memories, memory controllers, DMA, and so on, while in APB are attached all low-power low performance peripherals of the system, like UARTs, I/O, LCD controllers, etc. Both levels are interconnected by a AHB APB bridge.

AHB is a *multi-master multi-slave* bus, and therefore it relies on the existence of an arbiter and a decoder used to interpret the address of each transfer and provide a select signal (i.e. HSEL) for the slave that is involved in the transfer (see Figure 3.2(a) on the next page). Thus, a bus transaction is initiated by a bus master, which requests access from a central arbiter. The bus arbiter, if there are conflicts, ensures that only one bus master at a time is allowed to initiate data transfers. Even though the arbitration protocol is fixed, any arbitration algorithm, such as highest priority or fair access can be implemented depending on the application requirements. Once the arbiter issues a HGRANT to the master, the transfer starts. In AHB, transactions are pipelined to increase the bus effective bandwidth, and burst transactions are allowed to amortize the performance penalty of arbitration. Thus, all transactions are composed of an *address phase*, involving the address (i.e.

HADDR) and control wires (i.e. HTRANS, HSIZE, HBURST, HWRITE), and of a *data phase*, involving the separate data buses. There are two data buses, one for reads (i.e. HRDATA) and one for writes (i.e. HWDATA) are available, but only one of them can be active at any time.

To avoid bus starvation because of slow slaves and therefore poor bus utilization, AMBA provides two mechanism: *(i)* split/retry transfers, where high-latency slave can optionally suspend a bus transaction while its preparing the response, and *(ii)* early burst terminations, where if the arbiter detects too long transaction, it can release the burst in progress and grant another master. In addition, lock transfers also are supported by AMBA 2.0 AHB if a master requires exclusive access to a specific slave. However, multiple outstanding transactions are supported only to a very limited degree using split transactions, where a burst can be temporarily suspended (by a slave request). New bus transactions can be initiated during a split burst.



(a) AMBA 2.0 AHB shared bus (b) AMBA 2.0 AHB multi-layer

Figure 3.2: Block diagram of AMBA 2.0 AHB (bus vs. multi-layer)

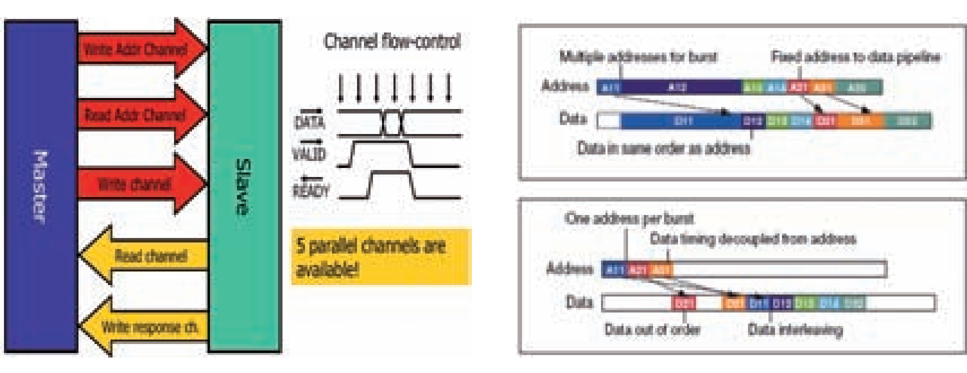
Despite this fact, AMBA 2.0 AHB can be also deployed on MPSoCs as a multi-layer AHB matrix (ML-AHB) or crossbar topology (see Figure 3.2(b)). Using this architecture as a backbone, multiple and parallel transactions can be done between an arbitrary master and slave on the system at the same time.

The ML-AHB busmatrix consists of input stage, decoder and output stage embedding arbiter. The input stage is responsible for holding the address and control information when the transfer to a slave is not able to commence immediately. The decoder determines which slave a transfer is

destined for. The output stage is used to select which of the various master input ports is routed to the slave. Each output stage has an arbiter. The arbiter looks at which input stage has to perform a transfer to the slave and decides that is currently the highest priority. Especially, the ML-AHB bus-matrix uses the slave-side arbitration scheme, i.e. the master just starts a transaction and waits for the slave response to proceed to the next transfer.

An evolution of AMBA 2.0 is *AMBA Advanced Extensible Interface (AXI)* [32]. AMBA AXI methodology brings a new step forward for high bandwidth and low latency design with backward AHB compatibility. Thus, AMBA AXI presents some useful pipelining as in AMBA 2.0 AHB, and it adds novel features that will benefit the design of larger chips requiring scalable interconnects at high frequency.

As shown in Figure 3.3(a), AMBA AXI is s a fully point-to-point fabric consisting in 5 unidirectional parallel channels: *(i)* read address channel, *(ii)* write address channel, *(iii)* write data channel *(iv)* read data channel, and *(v)* response channel. All channels are fully decoupled in time, which enable optimization, by breaking timing paths as required to maximize clock frequency and minimize latency.



(a) AXI Point-to-Point scheme (b) AMBA 2.0 AHB vs. AXI transactions

Figure 3.3: AXI Overview, and AXI vs. AMBA 2.0 AHB transactions

With symmetrical or point-to-point master and slave interface, AXI can be easily used anything from simple directed point-to-point architecture to multi-hierarchy systems by reducing the gate count on handshakes due to the simplicity of point-to-point links and hence timing penalty when signals traverse complex SoCs. Any master and slave of the system is completely agnostic from the interconnection, without the necessity to use arbitration or decoder modules as in AHB. This approach allows seamless topology scaling compared with a simple AMBA 2.0 AHB hierarchical shared-bus backbone.

In AXI, a transaction implies activity in multiple channels. For instance, a read transaction uses the read address and read channel. Because each channel acts independently, each transaction is labeled when start at the address channel. Thus, it supports for multiple outstanding transactions (in order like AMBA 2.0 AHB, but also out-of-order delivery as shown in Figure 3.3(b) on the preceding page), enabling parallel execution of bursts, resulting in greater data throughput than in AMBA 2.0 AHB. Moreover, to minimize the protocol information single-address burst transactions generates multiple data on read/write channels. All these features facilitates both high-performance and better average use of the interconnection fabric, and enable low-power as tasks can complete in a short time.

3.3.2 OCP-IP Overview

The Open Core Protocol™ (OCP) defines a high-performance, bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs for SoC designs. OCP is a superset of VCI [11] additionally supporting configurable sideband control, and defines protocols to unify all of the inter-core communication. This protocol has in some features some similarities with AXI.

The OCP defines a point-to-point interface between two communicating entities and is completely independent from the interconnection fabric. OCP cores can be tested of any bus-independent interface by connecting a master to a slave without an intervening on chip bus. Thus, the use of decoders and arbiters are hidden from cores, and must be addressed by using a basic command field. Thus, this permits OCP compliant master to pass command fields across the OCP that the bus or the NoC interface logic can convert, for instance, into an arbitration request, decode command, etc.

To support concurrency and out-of-order processing of transfers, the extended OCP supports the notion of multiple threads. Transactions within different threads have no ordering requirements, and so can be processed out of order. Within a single thread of data flow, all OCP transfers must remain ordered.

OCP is based on two basic commands, Read and Write, but five command extensions are also supported (ReadEx, ReadLinked, WriteNonPost, WriteConditional, Broadcast). An interesting feature of OCP is that offers posted writes (Writes) or WriteNonPost, i.e. writes without or with response, respectively. The other commands are basically used form synchronization between initiators. ReadExclusive is paired with Write or WriteNonPost, and has blocking semantics. ReadLinked, used in conjunction with WriteConditional has non-blocking (lazy) semantics. These syn-

chronization primitives correspond to those available natively in the instruction sets of different processors.

OCP it also support pipelining and burst transfers. To support pipelining, the return of read data and the provision of write data may be delayed after the presentation of the associated request. On the other side, burst can either include addressing information for each successive command (which simplifies the requirements for address sequencing/burst count processing in the slave), or include addressing information only once for the entire burst.

An specific features that makes OCP flexible and extensible is the ability to pass core-specific through in-band information on requests and responses, as well as read and write data (e.g. interrupts, cacheable information, data parity, etc.)

3.4 Related work

NI design has received considerable attention for parallel computers [150–152] and computer networks [153,154]. These designs are optimized for performance (high throughput, low latency), and often consist of a dedicated processor, and large amount of buffering. As a consequence, their cost is too high to be applicable on chip.

In [155] different packetization strategies have been investigated: *(i)* software library, *(ii)* on-core module based and *(iii)* off-core wrapper based. Finally, they evaluate each approach mainly in terms of latency and area costs. The conclusion of this work is that an off-core wrapper based approach is the best solutions in terms of latency and area.

Thus, a large body of research works implements NIs. In [156] is presented a NoC architecture in which the router and the NI are integrated in a single block, called communication interface.

In [157] is proposed the NoC to be viewed as an interconnect intellectual property module. The NI of this NoC offers services at the transport layer, being responsible for message to/from packet conversion. An NI example offering a unidirectional 64-bit data only interface to a CPU is presented.

QNoC NI is explained in [87] offering bus-like protocol with conventional read and write semantics, but also they also propose four service levels: *(i)* signaling, *(ii)* real-time, *(iii)* read/write and *(iv)* block transfers.

A dual-NI is reported in [158]. In this work, the target is to improve NoC architecture to be fault tolerance. Some network topologies and routing algorithms are presented to maximize the packet delivering packet to all cores in case of a faulty on the topology.

According to socket OCP-IP standard, in [159, 160] are reported an OCP compliant used on the MANGO clockless NoC architecture. A similar work is presented in [52, 53] on top of ×pipes NoC library. In addition, very close to our work in [161, 162], AMBA 2.0 NIs are presented to interconnect AMBA-based IP cores. However, all these works lack of compatibility and interoperability to design heterogeneous multi-protocol system.

The related work on multi-protocol NIs design is not very common at research level. It is probably because is more an engineering task rather than a long term research work, but from the point of view of the author is definitely a key challenge to ensure backward compatibility and coexistence of different standard protocols, as well as to enable IP core reuse, but also to enable NoC services at NI level.

Thus, in [163] the author exposes and presents the idea of VC-neutral NoC design. This works introduce the necessity and potential issues at the moment of design full-featured NIs (named NIUs in this work) which are able to communicate different standard protocols, such as AMBA 2.0, AXI, VCI and OCP. In fact, nowadays, it seems most of the NoC companies, such as Arteris S.A. [110], iNoCs SaRL [111], Sonics Inc [113] claim to have NIs capable to interact with several protocols, but it is unclear the grade of interoperability between them.

An interesting work of Philips Research Laboratories is reported in [164]. A complete NI compatible with multiple protocols standards, such as AXI, OCP and DTL [165] is presented, providing both guaranteed and best effort services via connections on top of Æthereal NoC. This services are configured via NI ports using the network itself, instead of a separate control interconnect.

The existing NI designs address one or more of the aspects of our NI, like low cost, high-level NoC services offering an abstraction of the NoC, and plug&play modular design and integration. However, in this work we present full-featured AMBA 2.0 AHB NI fully compatible with OCP transactions. The NI design is targeted to be extremely slim (hundred LUTs) and very low-latency. Furthermore, in this work we will take into account the inherent incompatibilities of both standards (i.e. byte enable, alignments, protection, burst types, etc).

Later, both NIs, AMBA 2.0 AHB and OCP will be extended to enable software programmability of different QoS features at NoC-level (i.e. guaranteed and best effort services). The outcome will be a complete solution to configure NoC features through IP cores at runtime (see Chapter 4 on page 85).

3.5 Full-Featured AMBA 2.0 AHB NI for ×pipes NoC

As explained before in Section 3.3 on page 60, AMBA 2.0 (AHB-APB) is an hierarchical bus interconnected with a AHB-APB bridge. In this section we show the design of a full-featured AMBA 2.0 AHB NI, targeted to translate AHB transactions to NoC packets and flits.

The NI back-end works up to the transport layer (see Figure 2.7 on page 33) injecting/absorbing the flits leaving/arriving at the IP core, whereas the front-end is responsible to pack/unpack the AHB signals coming from or reaching AHB compatible IP cores consisting in multiple message/flits. In other words, the NI is in charge to convert multiple transactions (i.e. requests/responses packets) into network flits and vice versa by adapting the clock domains.

In this work, the goal is to design a completely renewed AMBA 2.0 AHB NI fully compatible with OCP at transactional level. As shown in Figure 3.4 on the following page, the basic block of our design will be divided in two parts:

(i) AHB NI Initiator: slave module connected to each master (e.g. mainly processors, like CPUs, VLIWs, DSPs, Image Processing ASIPs, etc) of the system.

(ii) AHB NI Target: master module attached to each slave (e.g. memory, such SRAM or DRAM, but also custom co-processors accelerators, such as FPUs, FFT, etc) of the system.

×pipes leverages static source routing, which means that a dedicated *Look-up Table (LUT)* is present in each NI to specify the path that packets will follow in the network to reach their final destination. This type of routing minimizes the complexity of the routing logic in the switches. Furthermore, NIs also optionally provide buffering resources to improve performance.

Finally, as shown in Figure 3.4 on the next page, two different clock signals are connected to our AMBA 2.0 AHB NI: one coming from AHB interface towards NI front-end (i.e. `HCLK`), and another (i.e. `xpipes_clk`) to drive the NI back-end which is used by ×pipes protocol. For simplicity, the back-end clock must, however, have a frequency which is an integer multiple of that of the front-end clock. This arrangement allows the NoC to run faster even though some of the attached IP cores are slower, which is crucial to keep transaction latency low. The constraint on integer divider ratios is

instrumental in reducing the implementation cost of this facility down to almost zero, but decrease the design possibilities. However, a mesochronous or a GALS scheme can be designed effortlessly extending the NoC [57,166]. Therefore, each IP core can run at different frequencies of the ×pipes frequency, mixed-clock platforms are possible.

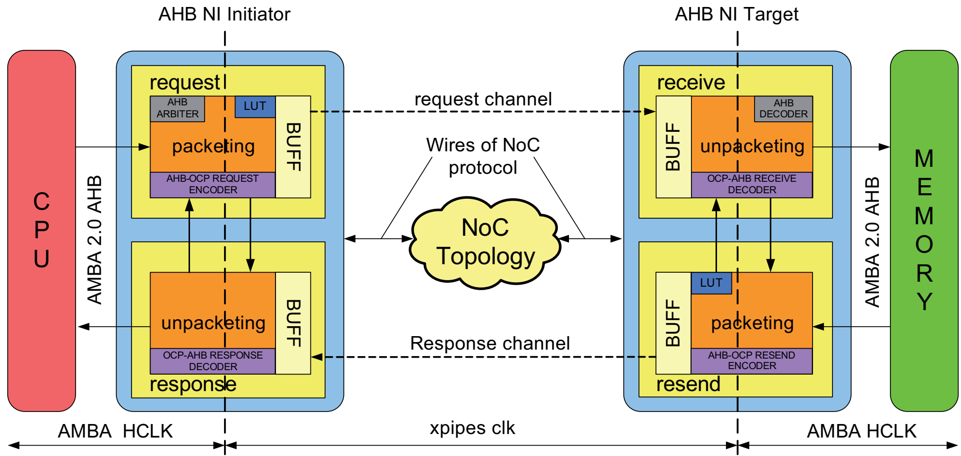


Figure 3.4: AHB NI block diagram in ×pipes NoC

3.5.1 Packet Definition and Extension

The packet format is the most important piece on in order to achieve backward compatibility between different protocols, and to design a full-featured AHB NI compliant. In this case of study, we define an extended or abstract packet format based on OCP codification but suitable to place all AHB signals coded in a proper way. As shown in Figure 3.4, to enable full-featured (i.e. read/write single, precise or imprecise burst transactions, etc) compatibility between AHB and OCP, in our design we include pairs of encoders/decoders in each AHB NI, to adapt AHB codifications to OCP.

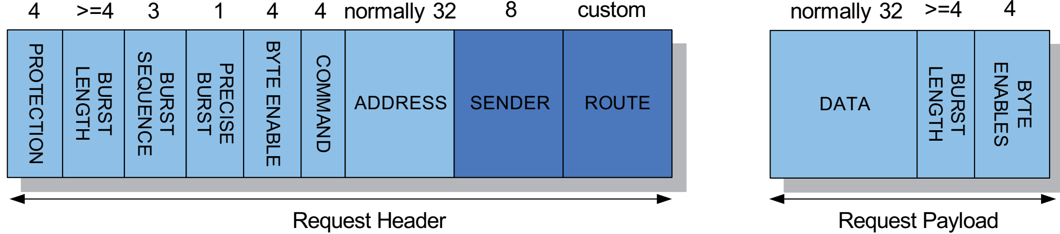


Figure 3.5: AMBA 2.0 AHB – OCP request packet

Different types of packets are defined by the virtual paths between NI blocks. We distinct two kind of packets: *request packet* (see Figure 3.5 on the preceding page) which goes in direction from master to slave, and *response packet* (see Figure 3.6 on the following page) which goes in the opposite direction, from the slave to the master. Moreover, each packet is subdivided in header and payload.

As shown in Figure 3.5 on the preceding page, in request header we place all AHB control signals codified them in a proper way to be compatible with OCP. In Table 3.1, we show the coding of each field in the request header according to AHB and OCP.

Field	Codification – Signal Mapping
Protection	Encodes protection (i.e. `HPROT`) present in AHB specification No equivalence in OCP[1]
Burst Length	Is a numeric value expressing the burst length. Encodes `HBURST`[2] (AHB) & `MBurstLength` (OCP)
Burst Sequence	**000 (INCR)** ↔ if `HBURST == SINGLE/INCR`[3] (AHB) or if `MBurstSeq == INCR` (OCP) **001 (WRAP)** ↔ if `HBURST == WRAP` (AHB) or if `MBurstSeq == WRAP` (OCP) **101 (STRM)** ↔ if `MBurstSeq == STRM` (OCP), No equivalent in AHB
Burst Precise	**0** ↔ if `HBURST != INCR` (AHB) **1** ↔ if `HBURST == INCR` (AHB) Simply encodes MBurstPrecise in OCP
Byte Enable	One bit per byte in the data field of the payload Encodes `HSIZE` (AHB) & `MByteEn` (OCP)
Command	**001 (WRITE)** ↔ if `HWRITE == True` (AHB) or `MCmd == WR` (OCP) **010 (READ)** ↔ if `WRITE == False` (AHB) or `MCmd == RD` (OCP) **101 (WRNP)** ↔ if `MCmd == WRNP` (OCP), no equivalent (AHB) **Others** ↔ reserved for future usage
Address[5]	Encodes `HADDR` (AHB) or `Maddr` (OCP)

[1] There is no equivalence in OCP, and therefore this field is simply ignored when it is received on an OCP NI Target.

[2] For `HBURST` equal to INCR4/INCR8/INCR16 or WRAP4/WRAP8/WRAP6 this field codifies the numeric value, i.e. 4, 8, 16.

[3] In AHB, it is possible to define single read/write requests by doing a undefined-length burst (INCR) of burst length equal to 1.

[4] Some byte enables OCP semantics (such as are not supported on AHB standard.

[5] The most significant byte of the address filed is used to refer a specific slave on the NoC.

Table 3.1: Mapping AHB – OCP signals on the request header

The last two fields of request header, *sender* and *route*, are completely independent from any protocol. The sender is an specific ID (of 8 bits normally) to identify the requested source, which allows to integrate at least 256 cores on the network, whereas the route, is a variable length field which determines the path of the packet along the NoC. The length of this field is determined by Equation 3.1.

$$route_{width} = log_2(N) \times M \quad (3.1)$$

where N is the maximum number of ports of all switches on the NoC, and M is the maxim number of hops that a packet can do to traverse the NoC.

On the request payload, the mapping of the signals of each protocol is straightforward. On the *DATA* field, only if it is a write requests, we encode the data to be written, i.e. HWDATA (AHB) or MData (OCP), otherwise this field is removed for optimization. In *Burst Length* field we encode as before a numeric value expressing the burst length. However, both AHB and OCP allow undefined-length bursts, so that this field will encode a mix of HBURST and HTRANS (AHB), or MBurstLength (OCP). Finally, as in the request header, the *Byte Enable* field encodes HSIZE or MByteEn but this one overrides the one provided in the request header to allow different byte enable alignments on each piece of data during transactions.

On the response packet (see Figure 3.6), AHB & OCP signals are mapped effortlessly. The header fields has the same meaning as in request packet, but in the opposite direction, i.e. changing the role of the sender, and including the reverse path on the route field. On the *DATA* field, it is encoded HRDATA (AHB), and SData (OCP) which is data coming from the slave under a read transaction. Finally, in *RESP* field we encode HRESP (AHB) or SResp (OCP). This field is very important because provides additional information on the status of the transaction. AHB and OCP encodes four types of responses, but in this work, we only support *OKAY/DVA* and *ERR*, since exclusive access (SPLIT/RETRY) can be supported using QoS at NoC level.

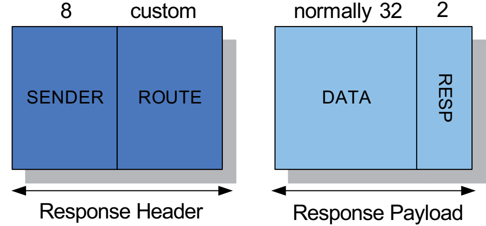


Figure 3.6: AMBA 2.0 AHB – OCP response packet

Once these request and response packets are composed and properly codified, they are then split into a sequence of flits before transmission, to decrease the physical wire parallelism requirements. The width of the flits in ×pipes is a fully configurable parameter; depending on the needs, ×pipes designs can have as few as 4 wires carrying data, or as many as in a highly parallel bus (64-bit buses typically have close to 200 wires, considering a read bus, a write bus, an address bus, and several control lines).

To provide complete deployment flexibility, in addition to being parameterizable in terms of flit width, the designed AHB NI is also parameterizable in the width of each packet fields. Depending on the resulting ratios, a variable amount of flits will be needed to encode an AHB or an OCP transaction. However, once we know the size of each field, the offsets are fixed, so that the packeting and unpacketing logic can operate at high frequencies.

3.5.2 AMBA 2.0 AHB Transactions to NoC Flits

In this section, it is explained in detail how to the NIs translate AHB transaction to NoC flits and vice versa. Two types of NIs are required, *(i)* AHB NI Initiator and *(ii)* AHB NI target to interact with master and slave IP cores, respectively. However, as a part of this main modules, some submodules must be designed in charge to pack/unpack request response packets from/to the NoC:

(i) ahb_ni_request: pack AHB transactions coming from an AHB master (e.g. CPU) using the request packet format, and at the same time, the packet is split in a stream of flits to be transmitted over the NoC.

(ii) ahb_ni_receive: unpack the transaction coming to the target NI from the NoC by a sequence of one or more flits, and afterwards, it regenerates the transaction in order to interact with an arbitrary AHB slave IP core (e.g. memory) on the system.

(iii) ahb_ni_resend: pack the AHB response coming from the AHB slave (e.g. memory) following the request packet format in a sequence of flits to be forwarded back to the master that did the request.

(iv) ahb_ni_response: unpack NoC flits arriving from the NoC, creates the response transaction to be delivered to the AHB master (e.g. CPU). Once this module finishes, it notifies ahb_ni_request that another transaction can be issued.

As explained before in Section 3.3 on page 60, AMBA 2.0 AHB is a multi-master multi-slave bus-based fabric architecture based on request and response transactions. Moreover, AHB relies on the existence of a shared bus which includes a centralized arbiter and decoder (see Figure 3.2(a) on page 62). The decoder is used to select a specific slave using the address issued by the master, whereas the arbiter is in charge to prioritize a master in case multiple request are issued on the bus simultaneously.

Therefore, the interface among IP cores and NIs can not be defined as in AHB, where the masters requests the arbiter and afterwards the decoder

selects the corresponding slave. As shown in Figure 3.4 on page 68, the proposed AHB NI Initiator in order to be fully AMBA 2.0 AHB compliant include the typical arbitration cycles described in the specifications [14, 15]. On the other side, the AHB NI target integrates the decoder logic to select the specific slave. These logic must be included in a distributed way on the FSM of *ahb_ni_request* and *ahb_ni_receive*, respectively. This does not happen when we design NI with point-to-point protocols, such as OCP, AXI, because they do not rely on the existence of this shared-bus specific modules present in AHB.

To translate or serialize AHB transactions to NoC flits, and vice versa, in each of these submodules presented before includes a quite complex FSM[2]. In the next sections, we present the FSMs of each submodule.

AHB FSM Request

The *ahb_ni_request* FSM is a module that packs AHB request transfers, serializing and converting them into a sequence of flits to be transmitted over the NoC. As shown in Figure 3.7 on the next page, the FSM in the beginning remains in *IDLE* until the master request to initiate a transaction. When a master IP core want to issue a new transaction, it request the bus, and if the NI is ready, i.e. there is not an ongoing flit (i.e. `!tx_gone`), or a response in progress (i.e. `!full_response`), it asserts `HGRANT`, otherwise it will remain until the previous transaction finish. Once these arbitration cycles are done, the transaction itself can be initiated, and therefore, the FSM will jump to the *START_TRANSACTION* state[3].

Once the transaction can start, we distinct between all multiple types of transactions that can occur in AHB:

(i) Single Write: in this type of transaction only one header and one payload request with the data to be written is transmitted over the NoC.

(ii) Precise Burst Write: a header and multiple payload request according burst length fields are transmitted over the NoC.

(iii) Imprecise Burst Write: as in precise burst write, a header and multiple payload request are transmitted over the NoC.

[2] Each FSM is not presented in fully detail due to its complexity (for instance, *ahb_receive_fsm* has 16 states, ten of transitions, 12 inputs and 12 outputs), and therefore, only states and transitions are reported in order to understand the general behaviour.

[3] AHB Lite modules can jump directly to this phase since they do not need the arbitration phase, but also can be adapted using an AHB Lite to AHB wrapper to be connected to the NI.

(iv) Imprecise Burst Read: as in precise burst write, a header and multiple payload request are transmitted, but now any data is included in the payload request, only the control information.

(v) Single or Precise Burst Read: on both cases only a header is transmitted over the NoC. The information on header request is enough to reconstruct the transaction on *ahb_ni_receive* module. This mechanism is also present in OCP and AXI, and in our case it leads to save bandwidth on the NoC.

Figure 3.7: AMBA 2.0 AHB NI request FSM

In all these states, the FSM is monitoring constantly `tx_gone` (an internal control signal that is asserted when a header or payload is ready to be transmitted), and `HTRANS` (AHB signal that contains the information about the status of the transfer) to know exactly which is the next state.

Two output signals must be explained in order to understand the handshaking process between the AHB master, the *ahb_ni_request* and the NoC, `HREADY` and `flit_sequence`.

HREADY will be asserted on every transfer of each AHB address or data phase when all the valuable information on the bus is sampled and processed by the FSM in order to notify the master when the transfer is finished. Furthermore, when a busy transfer occurs, i.e. `HTRANS==BUSY`,

the FSM suppress this transaction asserting HREADY but without generating any overhead traffic on the NoC.

At the same time, the FSM marks the flit_sequence as *header flit* whether the FSM is going to start a transaction. After that, while the FSM is processing the packets and jumping over the different states, flit_sequence is tagged as a *body flit*. Finally, when the last flit is going out, flit_sequence is marked with a *tail flit*. As shown in Figure 3.7 on the preceding page, at the end of each final phase of each type of transaction, when the lasts flits are ready to be transmitted (i.e. tx_gone is high), the FSM can jump to *IDLE*, or to *BUS_GRANT* if the master request another transaction (i.e. HBUSREQ is high) by the master. This mechanism pipelines the end of the transaction with the beginning of the next one saving one clock cycle each time, and improving the injection rate on the NoC.

AHB FSM Receive

The *ahb_ni_receive* FSM module (see Figure 3.8 on the next page), it is probably the most complex because of its synchronization requirements, specially for read transactions. This module must deal and synchronize properly several modules at the same time. It must receive flits from the NoC, unpack them into a request header, and one or more request payload packets (depending if the request is a single or burst transaction). Simultaneously, it should re-generate them in a pipeline fashion according to AHB specifications [14].

In addition, it must perform the handshake of each address and data phase with the AHB slave using HREADYin and HREADYout, and in case the transfer is a read, it has to exchange control (i.e. is_resending) information with *ahb_ni_resend* module in order to know if each piece of the read transfer have been resent successfully.

The receive FSM is in IDLE state until it receives the full header of a AHB request transfer and there is not any previous transfer to be resent (i.e. full_header && !is_resending). Once the transfer is ready to be initiated could happen that the last transfer of the previous read transaction is still ongoing (because of congestion in the network, slow slave, etc), and therefore, the FSM can not issue a new request transfer (i.e. !enable_new_request). In that case, the FSM jumps to *WAITING* state.

When a transfer is ready to be initiated (i.e. full_header && !is_resending && enable_new_request) depending the received command, the burst sequence and if it precise or not, the FSM has to process between different type of transactions: *(i)* precise read, *(ii)* imprecise read, *(iii)* precise write and *(iv)* imprecise burst.

Figure 3.8: AMBA 2.0 AHB NI receive FSM

When a transaction is received and it is going to start, the FSM automatically generates HSEL signal, which is in charge to select the corresponding AHB slave.

For single or precise read burst request only the header is sent from *ahb_ni_request* to *ahb_ni_receive*. Each time that an address phase or data phase is processed successfully (i.e. HREADYin[4] is high), an address generator increments the address sequence according HSIZE and HBURST. For instance, during a 32-bits 4-burst beat transaction, this module creates a relative offset addresses of 0x0, 0x4, 0x8 and 0xC.

Concurrently, each time a data phase is processed by the FSM, an internal counter (i.e. burst_counter) is incremented in order to provide a stopping mark when a read/write precise burst is going to finish. The FSM uses this stopping mark in conjunction with full_payload and packet_finish in order to identify where single, precise or undefined-length burst transactions ends.

Every time the FSM module has a transfer ready (address or data phases)

[4] HREADYin is a FSM input coming from the AHB slave, in other words, is the HREADYout of the slave. The slave raise his HREADYout to notify the master, in this case *ahb_ni_receive_fsm* within AHB NI Target, each time that a response is ready.

to be processed, it notifies the slave raising HREADYout[5].

On the other side, if the transfer is not ready at the AHB NI Target due to congestion on the NoC, busy transfers are generated (i.e. HTRANS==BUSY) by the FSM to notify the slave that the transfer is going to be delayed. Under normal conditions, the FSM generates AHB pipeline transactions putting on HTRANS the typical values, i.e. NONSEQ, SEQ, SEQ,...and then NONSEQ or IDLE to start again the next transaction. The FSM also generates a mask/unmask control signals for command and data phases which are asserted properly to enable all output AHB signals over the bus to the slave, when all packet flits have been unpacked and processed.

Once the transaction completes[6] the FSM resets both, the address generator and the burst counter to be ready for the next AHB transaction.

AHB FSM Resend

The *ahb_ni_resend* FSM takes the AHB responses from the AHB IP core slave, and create the response packet which is split in a sequence of flits over the NoC. As shown Figure 3.9, the FSM is in *IDLE* until a new response arrive, i.e. start_receive_response is high. Then, the FSM waits to fill the header and payload (one or multiple depending if the request is a single or burst read) of the packet response.

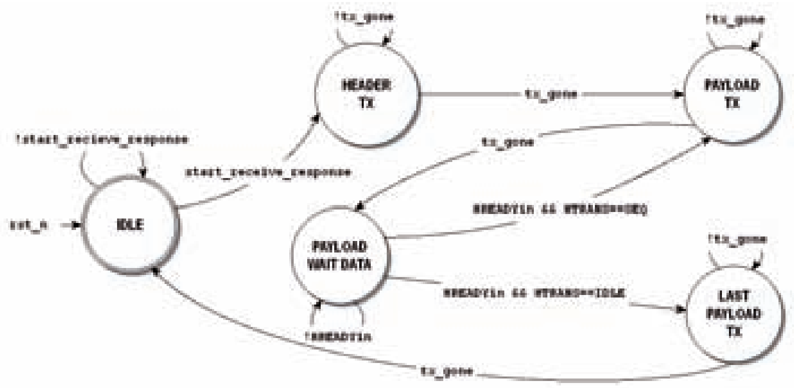


Figure 3.9: AMBA 2.0 AHB NI resend FSM

[5]HREADYout is a FSM output from AHB NI Target over the AHB slave, or in other words, is the HREADYin of the slave. On reset conditions, the FSM assign HREADYout to high to enable the slave to be ready to sample an AHB access

[6]For simplicity, and to avoid drawing more lines from states where the transaction finishes to the initial *IDLE* state, in Figure 3.8 on the preceding page, we shown two initial/final states labeled as IDLE, but it is important to remark that only one is present in the FSM and, both has exactly the same meaning.

In *HEADER_TX* and *PAYLOAD_TX* states, AHB control and data signals are sampled, and at the same time, the FSM sends flits out over the NoC. If an error transfer occurs, i.e. `HRESP==ERR`, the FSM ignores this transfer and no flits are transmitted over the network.

When the transaction starts, in *HEADER_TX*, we mark the flits as *header flit*, then in all the other states they are marked as *body flit*, and in the last state, i.e. *LAST_PAYLOAD_TX* the last flit is marked as a *tail flit* in order to close the wormhole packet.

From AHB slave, response transfers can be extended or delayed one or more clock cycles whether `HREADY` signal from the slave to *ahb_ni_resend* module. This often occurs when slow slaves require few cycles to trigger the response because they are calculating the output data (i.e. FPU, FFT, JPEG, etc). To control this behaviour, a *PAYLOAD_WAIT_DATA* state must be added. The FSM are looping here until, `HREADY` is high, which means that `HRDATA` is ready to be sampled and afterwards transmitted in a sequence for flits.

AHB FSM Response

The *ahb_ni_response* FSM is in charge to unpack AHB responses coming from AHB NI Target, concretely coming from *ahb_ni_resend* module through the NoC on a sequence of flits. In Figure 3.10, we show that the FSM is in *IDLE* until `response_awaited` is high, or in other words, waiting until a new response must be processed.

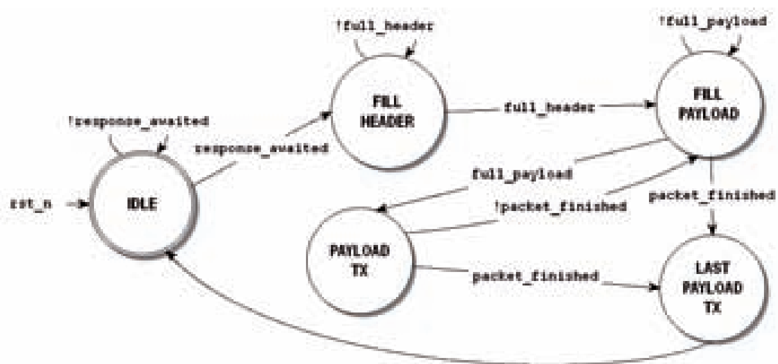


Figure 3.10: AMBA 2.0 AHB NI response FSM

Afterwards, it unpacks the header using the control line `full_header`, which indicate that all the flits of the header have been received. Once the header is valid and processed, the FSM is doing the same with each payload

(one or more depending whether the response is a single or a burst read) using `full_payload`. Every time that full payload is received, the FSM transmits its response back to the AHB master through `HRDATA` and `HRESP` signals.

This FSM has an output to mask/unmask (i.e. `mask_response`) the AHB pinout, enabling or disabling the AHB pinout whenever it is required. This allows to wait until all flits of each payload are received completely and the response is stable, and therefore, it can be transferred over the AHB master.

3.6 Experimental results

To assess the validity of our AMBA 2.0 AHB NI, we employ a full-functional verification to ensure the reliability of a transparent traffic communication at transactional level between AMBA and OCP IP cores on custom NoC architecture. Moreover, we employ hardware synthesis on different FPGAs platforms to quantify the area costs of our AMBA 2.0 AHB NI against a OCP-NI.

3.6.1 Functional Verification on a AMBA-OCP Start NoC

To face a full-functional verification of our AMBA 2.0 AHB NI, and its compatibility with OCP-IP transactions, we mainly employ two communication patterns between AMBA and OCP cores:

(i) AMBA – AMBA communication.

(ii) AMBA – OCP / OCP – AMBA communication.

To test this communication patterns a simple heterogeneous NoC-based architecture with 2 cores as initiators (see Figure 3.11 on the next page) has been developed:

(i) AMBA 2.0 AHB compliant traffic generator (AMBA TGEN).

(ii) AMBA 2.0 AHB memory.

(iii) OCP traffic compliant traffic generator (OCP TGEN).

(iv) OCP memory.

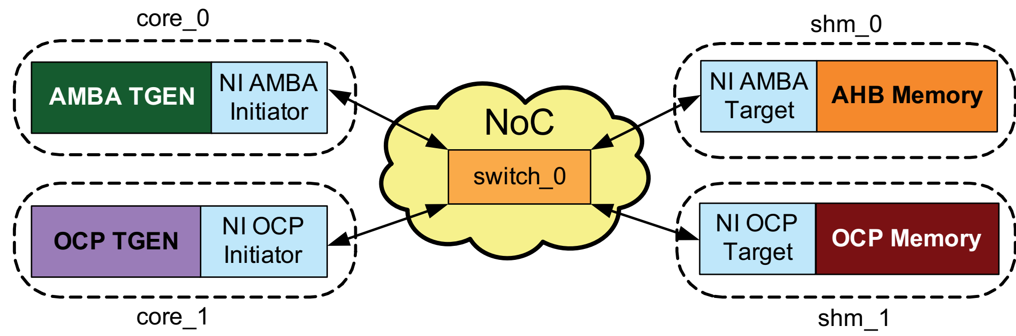


Figure 3.11: Single switch AMBA 2.0 AHB – OCP star topology

To describe this topology, we use ×pipescompiler [69] and the topology description file presented in Listing 3.1. Both AMBA and OCP memories have been specified as a shared memories, and there is a route to/from OCP and AMBA traffic generators on the system in order to check AMBA-OCP traffic compatibility, as well as among pure AMBA NoC-based systems.

A fully customizable AMBA 2.0 AHB components (i.e. AMBA TGEN, and AHB memory) have been developed from scratch. The AMBA 2.0 AHB traffic generator is able to run any type of AHB transactions:

- Single transactions (SINGLE).
- Incremental bursts (INCR4/INCR8/INCR16).
- Wrapping bursts (WRAP4/WRAP8/WRAP16).
- Undefined-length bursts (INCR).

The AHB memory can also be parameterized, in terms of word size and depths, as well as different independent write/read latencies, and it is able to process different types of single or burst transactions.

Using this highly customizable framework an exhaustive verification test have been done using the developed AMBA 2.0 AHB complaint traffic generator. This module generate a set of consecutive write and read using all possible types of transactions between the AMBA TGEN cores and the AHB memory (placed at 0x00), but also between AMBA TGEN and OCP memory core (placed at 0x01), and vice versa. Notice, that the 8 MSB bits are used to address any slave IP core on the NoC-based system.

The idea to generate pairs of write and read transaction is to verify the content of the memory with a read after the write transaction is finished. Checking the integrity of the data on the memory will ensure the correct functionality at transactional level of the AMBA 2.0 AHB NI.

If an error occur during the verification process, an assertion is raised, an a debug signal indicates the error.

Listing 3.1: AMBA-OCP topology description file

```
// ---------------------------------------------------------------------------
// Define the topology here
// name, mesh/torus specifier ("mesh"/"torus"/"other")
// ---------------------------------------------------------------------------
topology(4x4_AMBA_OCP, other);

//---------------------------------------------------------------------------
// Define the cores
// core name, NI clock divider, input buffer depth, output buffer depth,
// type of core ("initiator"/"target"), memory mapping (only if target),
// "fixed" specifier (only if target and of shared type),
// NI type (amba, ocp)
//---------------------------------------------------------------------------
core(core_0,  1, 2, 6, tester, initiator, amba);
core(core_1,  1, 2, 6, tester, initiator, ocp);

core(shm_0,   1, 2, 6, tester, target:0x00-fixed, amba);
core(shm_1,   1, 2, 6, tester, target:0x01-fixed, ocp);

// ---------------------------------------------------------------------------
// Define the switches here
// One switch of 4x4 ports with 2 input and 6 output buffers
// ---------------------------------------------------------------------------
switch(switch_0, 4, 4, 2, 6);

// ---------------------------------------------------------------------------
// Define the links here
// link name, source, destination, number of pipeline stages (optional)
// ---------------------------------------------------------------------------
link(link_0,  core_0,    switch_0);
link(link_1,  switch_0,  core_0);
link(link_2,  core_1,    switch_0);
link(link_3,  switch_0,  core_1);
link(link_4,  shm_0,     switch_0);
link(link_5,  switch_0,  shm_0);
link(link_6,  shm_1,     switch_0);
link(link_7,  switch_0,  shm_1);

// ---------------------------------------------------------------------------
// Define the routes here
// source core, destination core, the order in which switches need to be
// traversed from the source core to the destination core
// ---------------------------------------------------------------------------
route(core_0, shm_0,    switches:0);
route(core_0, shm_1,    switches:0);
route(core_1, shm_0,    switches:0);
route(core_1, shm_1,    switches:0);

route(shm_0,  core_0,   switches:0);
route(shm_0,  core_1,   switches:0);
route(shm_1,  core_0,   switches:0);
route(shm_1,  core_1,   switches:0);
```

The tests have been done using a single clock domain, i.e. the NoCs and the NIs running at the same clock frequency, but also under a multiple

clock system using the integer clock divider present on the NI to validate the system behaviour under different clock domains (e.g. running HCLK at half of the ×pipes clock).

3.6.2 Synthesis of AMBA 2.0 AHB NI vs. OCP NI

In this section, we quantify the results by synthesizing our AMBA 2.0 AHB NI, and comparing it with the equivalent OCP NI design already present in ×pipes library. In these experiments we synthesize each module of our AHB NI in different FPGAs in order to extract a more representative results on different technologies.

Moreover, to do a fair comparison, both NIs, AMBA 2.0 AHB and OCP have been synthesized using the same parameters:

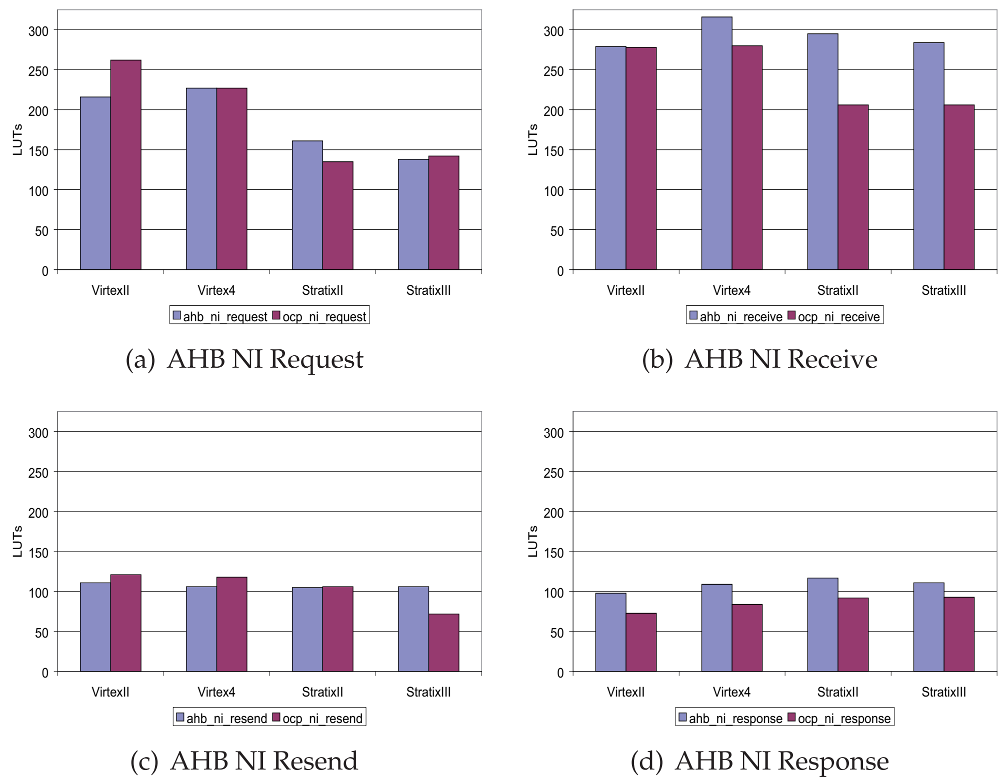
- 32-bits for address and write/read data buses.
- Minimum buffer capabilities (2 flits input buffers).
- 32-bits flit width.
- The ×pipes NoC use wormhole packet switching and stall&go flow control.

The results[7] shown in Figure 3.12 on the following page, demonstrate that our AMBA 2.0 AHB NI is more or less equivalent against the OCP NI in terms of area costs. Actually, this is not a fair comparison, because as explained before, our AMBA 2.0 AHB NI includes several encoders/decoders to encode properly AHB transactions to be compatible with OCP traffic. Obviously, this area overhead is not present in OCP NI since it is not necessary to perform any codification of any signal.

Detailing a bit more the results, in Figures 3.12(a), 3.12(c) on the next page, we show the results of synthesizing our *ahb_ni_request* and *ahb_ni_resend* modules in our set of FPGAs. It is easy to observe that, the area costs (taking the average of the synthesis in the different FPGAs) in LUTs are practically the same as OCP modules.

Nevertheless, when we compare *ahb_ni_receive* and *ahb_ni_response* against its equivalent OCP modules (see Figures 3.12(b), 3.12(d) on the following page) the results are not so similar. The *ahb_ni_receive* is 23.57% bigger than *ocp_ni_receive*, mainly because of the complexity of the FSM (at the

[7] The results have been extracted using Synplify® Premier 9.4 [167] to synthesize each component on different FPGAs without any optimization. VirtexII (xc2v1000bg575-6) and Virtex4 (xc4vfx140ff1517-11) from Xilinx, and StratixII (EP2S180F1020C4) and StratixIII (EP3SE110F1152C2) from Altera.



(a) AHB NI Request

(b) AHB NI Receive

(c) AHB NI Resend

(d) AHB NI Response

Figure 3.12: Synthesis of AMBA 2.0 AHB NI vs. OCP NI

time of re-generation pipeline AHB transactions), the longer request header packet (`HPROT` is not present in OCP systems), and the decoder to translate `HWRITE`, `HSIZE`, `HBURST`, `HSIZE` to OCP equivalent codes.

The 27.63% overhead in area costs of *ahb_ni_response* is caused by the fact this module is extremely small, and the inclusion of the decoder to code the responses impacts a lot on the final size.

In any case, the design of the AMBA 2.0 AHB NI fully compatible with OCP is still slim and efficient. The area costs are about 300 LUTs for AHB NI Initiator, and 400 LUTs for AHB NI Target, approximately.

It is important to remark, that the area costs of our AMBA 2.0 AHB NI is similar as long as you do not take in consideration the area overhead of the internal encoders/decoders. Even taking them into account, still the overhead is not critical looking at the final area of a complex NoC-based system.

On the other side, the results of circuit performance (f_{max}) are shown in Figure 3.13 on the next page. On this figure, it is easy to observe that our AMBA 2.0 AHB NI can operate at high frequencies even in FPGAs plat-

forms. As expected, since the area costs are quite similar to OCP NI, the results in terms of f_{max} are negligible when both NIs are compared. This behaviour it is easy to observe in Figure 3.13, where lines are more or less parallel according to the equivalent pair of modules AMBA/OCP. The impact on the maximum achievable frequency is on average around 1.5%, negligible in a NoC system, where usually the clock frequency is normally limited by the switches. In any case, our AHB NI taking the worst case delay can operate up to 300 MHz in the new generation FPGAs (StratixIII).

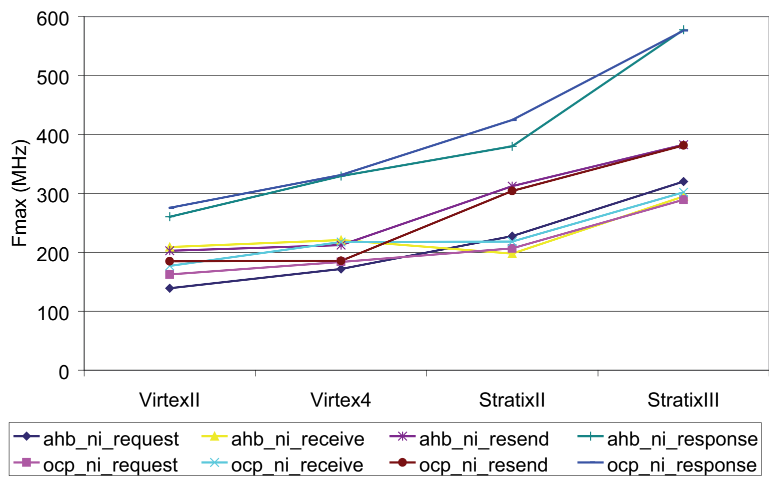


Figure 3.13: Circuit performance (f_{max}), AMBA 2.0 AHB vs. OCP NIs

3.7 Conclusion and Future Work

This work enables compatibility of AMBA 2.0 AHB and OCP transactions on the same NoC-based system. This allows seamless mixing of IP blocks fully compliant with these bus-based standard (i.e AMBA) and socket IP protocol (OCP). In other words, thanks to the NIs developed, we can mix AMBA and OCP IP cores and all types of transactions (i.e. INCR/WRAP/undefined-length burst types, multiple byte enables, request/response/resend and error protocols, etc) can coexist and interact between each other transparently depending on the application dynamism and the communication patterns. Clearly, this work illustrates many advantages form the point of view of the core.

The AHB NI have been implemented target to be slim and efficient, in terms of low-latency and high-performance due to the reduced area costs. It

optimizes the transmission efficiency and latency with an optimized packet format. The packets minimize the amount of information in each transmission that could instead be regenerated at the receiver (e.g. addresses during bursts).

This architectural support and backward compatibility leads to design highly heterogeneous NoC-based MPSoCs integrated by multi-protocol shared or hierarchical bus-based, or even multi-layer or crossbar subsystems. Inside these complex systems, due to the synchronization done in the NI FSM, multi-clock domains can work at the same time without losing data.

Further work can be done in many directions. Of course, the layered structure from transport to physical layer, and the flexibility of the modular design of the NI will allow effortlessly further extensions on the packet format to include more standard protocols, such as AXI [32], STBus [28] or Avalon [19] network interfaces.

To improve the actual work, we can add support for SPLIT and RETRY transactions, with are confined in the target NI, and support for LOCK transactions. However, these features can be done reusing QoS features present in NoC backbones. Thus, a master can block a channel during a period of time using circuit switching, and obviously, if another master request access to the same core, it will wait until the channel is released.

In terms of full compatibility between AHB and OCP, we can issue an error to handle the incompatibility of non-posted writes vs. posted writes, because in AMBA there is no way to request for posted transactions. The semantic differences among OCP and AMBA, in terms of byte enables and transaction alignments, and wrapping burst boundaries might be an interesting issue to solve. In the same direction, handling another level of heterogeneity with different core parameters (e.g. data width, address width, endiannes, etc.) also is a key point to improve this work. The automatic generation of *downsizers* or *upsizer* must be tackled by EDA tools, attaching this modules, if necessary, to each NI according to IP specs at design time. This core-centric feature will enlarge even more reuse, integrating heterogeneous cores at the moment to design NoC-based MPSoCs, enabling the transparent communication among the cores of the system.

CHAPTER 4

Runtime QoS support Through HW and Middleware Routines

This chapter[1] will introduce the extension of NIs and switches of the existing ×pipes NoC library in order to enable different types of QoS. Moreover, in this chapter are also introduces the low-level QoS-support routines to exploit the QoS features from the application at runtime. Finally the aim is to speed up real applications and synthetic benchmarks using this HW-SW infrastructure.

4.1 Motivation and Challenges

The emerging and modern heterogeneous SoCs or MPSoCs design consists of many different components and IP blocks (i.e. mix of general purpose processors, memories, DSPs, multimedia accelerators) interconnected by a NoC. These components can exhibit disparate traffic characteristics and constraints, such as requirements for guaranteed throughput and bounded communication latency. The problem is further compounded by the recent trend towards digital convergence, software-upgradeable devices and the need for dynamic power management. Chips are going to be used in various application scenarios, also called *use cases or scenarios* (for example, 3G connectivity as opposed to video gaming); their firmware and application code could be updated over the device's lifetime; and the various IP blocks may operate at different performance points. Each of these effects implies additional unpredictability of the on-chip traffic patterns, possibly render-

[1]The author will like to acknowledge contributions of all ×pipes development team but specially to Dr. Federico Angiolini

ing their static characterization impractical or overly conservative.

At design time, to provide certain QoS, designers with EDA tools often tend to overdimension the NoC backbone to fulfill the overall application requirements. However, this approach is suboptimal, and therefore, it should be merged with runtime QoS facilities to enable load balancing, prioritizing or exclusive access if the application requires it. It is, therefore, essential for a NoC system to support Quality of Service (QoS) through an efficient transport services while not resorting to resource overprovisioning, as is common in large area networks. Thus, different levels of QoS, i.e. *Best-Effort (BE)* and *Guaranteed Services (GS)*, should be available to applications, allowing designers to carefully allocated communication resources for high throughput or low latency traffic classes in a prioritized manner, but at a minimal power and area costs.

As an example, many designs nowadays require specific guarantees for particular data flows. For example, even if some of the chips can operate on a best-effort basis (results should be provided as quickly as possible but without a real performance bound - e.g. the time to browse an address book), some parts of it may require a stringent "guaranteed" requirement (results should be available within a maximum time - e.g. the time to play back the next video frame when watching a movie). The design of interconnects which mix BE and GS is a research challenge which is still very open.

In traditional networks [72,133] many techniques can be applied to provide QoS, but this scenario changes radically when designing NoC-based MPSoCs due to largely different objectives (e.g. nanosecond-level latencies), opportunities (e.g. more control on the software stack) and constraints (e.g. minimum buffering to minimize area and power costs). QoS for NoCs is impracticably implemented only in hardware, due to the large and hard-to-determine number of possible use cases at runtime; a proper mix of hardware facilities and software-controlled management policies is vital to achieving efficient results.

This facility must be controllable by the software stack to tolerate software updates over the lifetime of the chip, to make incremental optimizations possible even after tape-out, and to allow for different operating regimes as the use cases alternate at runtime.

Therefore, applications and/or the programming environment chosen for their development/execution should have some degree of control over the available NoC services. In this way, the application programmer or the software execution support layer (either an OS, a custom support middleware or runtime environment) can exploit the hardware resources so as to

meet critical requirements of the application and, more in general, speedup the execution while fitting a limited power budget. The access to the NoC configuration should be mediated by an easily usable API, should be low-overhead in terms of computation cycles, and should be compatible with mainstream multi-core programming approaches.

4.2 Related work

The NoC research community is focusing on a variety of design issues concerning heterogeneous NoC design, such as those presented in [58] (e.g. channel width, buffer size, application or IP mapping, routing algorithms, etc), as well as the design of Network Interfaces (NIs) for different bus-based protocols (e.g. AMBA/AXI [14, 32], OCP-IP [12]), adding virtual channels [98, 168] to improve the latency, designing efficient application-specific or custom topologies automatically [169], and bringing these designs towards different VLSI technologies, such as 65nm [170]. Despite, not a lot of effort has been made to study how to support low-cost QoS making it available at runtime across the software stack from application to packets.

Previous works to provide different levels of QoS at the hardware level are presented in [87, 135] where a complete NoC framework named QNoC is designed to split the NoC traffic in different QoS classes (i.e. signalling for inter-module control signals, real-time for delay-constrained bit streams, RD/WR that model short data access and block-transfers for handling large data bursts), but more work would be needed to expose these facilities across the software stack.

The work presented in [171] tries to combine BE and GT streams by handling them with a time-slot allocation mechanism in the NoC switches. A clockless NoC that mixes BE and Guaranteed Services (GS) by reserving virtual circuits is reported in [67, 91]. In addition, in [88, 172, 173] the authors claim to combine GT and BE in a efficient way, taking into account the required resource reservations for worst-case scenarios. Another generic approach following the idea offered in [87] about traffic classification is demonstrated in [79, 169, 174]. These works show a methodology to map multiple use-cases or traffic patterns onto a NoC architecture, satisfying the constraints of each use-case.

However, all these straightforward solutions are unfortunately expensive in terms of silicon requirements (area, power, frequency), while also being complicated to exploit at higher levels of abstraction (e.g. devising a set of use cases, accounting for application dynamism, etc.).

Very recently, the work presented in [175] introduce a method to manage

QoS at the task level on a NoC-based MPSoC, and in [176, 177], the authors report a detailed model on how to enable application-performance guarantees by applying dataflow graphs, as well as aelite, a NoC backbone based on Æthereal with composable and predictable services.

Our contribution is similar to these previous works, and but we focus on the actual viability of the hardware implementation by ensuring a reasonable hardware cost of QoS features at NoC level. Moreover, the work reported in [175] has been validated with a synthetic or custom traffic, while we leverage on a full functional MPSoC virtual platform [54] as we will show in Section 6.8 on page 166.

In this work, basically we are going to offered end-to-end QoS by:

- Best-Effort (BE): this services make no commitments about QoS. They refer to the basic connectivity with no guarantees. Such a service is provided using first-in, first-out (FIFO), or first-come, first-served (FCFS) scheduling based on *Fixed Priorities* or *Round Robin (RR)* arbitration scheme on the channels in each switch as in ×pipes, or even dynamically based on the NoC workload [178].

- Differentiated Services: splits the network traffic into several classes each with different requirements regarding data delivery. In this work a priority scheme (e.g. high or low priority packets) is implemented. Packets are treated according to the class they belong to. This QoS features is also known as soft QoS, because there is not hard guarantees are made for individual flows.

- Guaranteed Throughput (GT): this services ensure each flow has a negotiated bandwidth regardless of the behaviour of all other traffic. This type of QoS is performed by doing a circuit switching or TDMA technique on the channel in order to ensure end-to-end exclusivity access to the resources through reservation/release policy. Provides hard guarantees (e.g. real-time) in terms of deterministic latency and bandwidth on each specific flow.

In summary, our contribution is to implement low-level QoS features (i.e. a priority scheme and the allocation/releasing of end-to-end circuits) with a reasonable implementation cost. Then all QoS features will be exposed through the a lightweight low-level and middleware API (for differentiated services and GT traffic) across the software stack all the way up to OS layer, runtime environment, and application layers. This work enable the runtime provisioning of QoS to specific flows at application or task level, or event using QoS features through a global communicator or

scheduler/broker/supervisor which works at higher level, as is reported in [179, 180].

A similar approach to the actual work, in terms of middleware APIs, is reported in [181], where a NoC Assembler Language (NoC-AL) is proposed as an interface between NoC architectures and services, but mainly focus to support power management instead of runtime QoS.

4.3 NoC-level QoS Design

Homogeneous, but specially next generation heterogeneous NoC-based MPSoCs will run concurrently multiple applications according few use cases with different requirements, in terms of bandwidth and latency bounds. Hence, it is crucial to provide QoS features at NoC-level. Despite this fact, this work goes further, and it tries to enable this QoS present on the NoC backbone, as NoC services to the application or even on top of the parallel programming model.

We designed a low-cost NoC-level QoS features based on a priority scheme (to fit few use cases) in order to classify several types of traffic, and we also enable the capability of reserving channels to ensure end-to-end bandwidth or latency for certain traffic between specific IP cores on the NoC-based system. Thus, this work tries to support NoC services by combining BE and GT, as well as differentiated services, but without impacting a lot on the design of hardware components (in terms of area and power consumption).

In this work, we design NIs and switches to allow the capability of reserving channels or bandwidth for certain traffic or arbitrating packets with different priorities set by the application or the programming model.

Rather than a costly table-based TDMA [88, 103, 171] or virtual channel scheme [98], both QoS features have been designed thinking on its real hardware implementation or prototyping. Thus, a low cost priority scheme not based on virtual channels (instead parallel link can be deployed), and a guaranteed services based on channel reservation.

From the hardware point of view, a set of components have been redesigned using Verilog RTL to extract synthesis results, but also an equivalent SystemC™ RTL modules have been implemented to be included on MPARM [54] cycle accurate simulator through ×pipescompiler [69]. This will allow to explore the designed QoS in conjunction with software components (i.e. middleware routines, OS and the parallel programming model) using a virtual platform.

To disclose the QoS features at higher levels as NoC services, and to

enable runtime changes on the NoC backbone, we face a design at NoC level by extending the basic elements of ×pipes NoC library: *(i)* on the Network Interfaces (NIs) and *(ii)* within the switches.

4.3.1 Trigger QoS at Network Interfaces

The aim on design QoS features on the NIs is to expose them towards higher levels of abstraction. On NI, concretely on the NI initiator, the main target is to identify which type of QoS is requested by the processor, accelerator, etc, but also its programmability and reconfiguration at runtime.

On initiator NIs, a set of configuration registers, memory-mapped in the address space of the system are used to program the different levels of priority. These registers can be programmed to assign different levels of priority on each individual packet at runtime.

When the application demands a given QoS feature, the NI tags header packets in a suitable manner. To do this, an extension on the packet header presented in Section 3.5.1 on page 68 has been done. Figure 4.1(a) on page 92 shows the new packet format, and in Listing 4.1 we present the possible QoS encodings embedded on QoS field it, concretely, within the 4-bit QoS field.

Listing 4.1: QoS encoding on header packets

```
// Priority levels
#define ENC_QOS_HIGH_PRIORITY_PACKET_0    4'b0000
#define ENC_QOS_HIGH_PRIORITY_PACKET_1    4'b0001
#define ENC_QOS_HIGH_PRIORITY_PACKET_2    4'b0010
#define ENC_QOS_HIGH_PRIORITY_PACKET_3    4'b0011
#define ENC_QOS_HIGH_PRIORITY_PACKET_4    4'b0100
#define ENC_QOS_HIGH_PRIORITY_PACKET_5    4'b0101
#define ENC_QOS_HIGH_PRIORITY_PACKET_6    4'b0110
#define ENC_QOS_HIGH_PRIORITY_PACKET_7    4'b0111

// Open/close circuits
#define ENC_QOS_OPEN_CHANNEL              4'b1000
#define ENC_QOS_CLOSE_CHANNEL             4'b1001
```

The 4-bits of the QoS field let us to design until 8-level priority scheme to classify different types of traffic within the system. Thus, the level 0 has lowest priority, and level 7 is the most prioritized on the NoC architecture. To program a priority packet two phases must be performed; first, a NI programs the transaction to the specific configuration register with the priority level required, and afterwards, one or more real transactions or packets with the actual data. The actual priority scheme implementation is done in each switch depending on the channel request and the priority level embedded on the packet.

On the other side, to trigger the opening and closing of circuits in order

to block/release channels for a certain period of time, ”fake” transactions should be performed, in the beginning and in the end, in order to notify the switches. This mechanism is based on the emulation of a circuit switching on top of a wormhole packet switching scheme.

The normal behaviour to open/close an exclusive transmission/reception guaranteeing QoS is done in three phases:

(1) Open the circuit (unidirectional or bidirectional ”fake” transaction).

(2) Perform a normal stream of transactions or packets over the NoC under GT conditions.

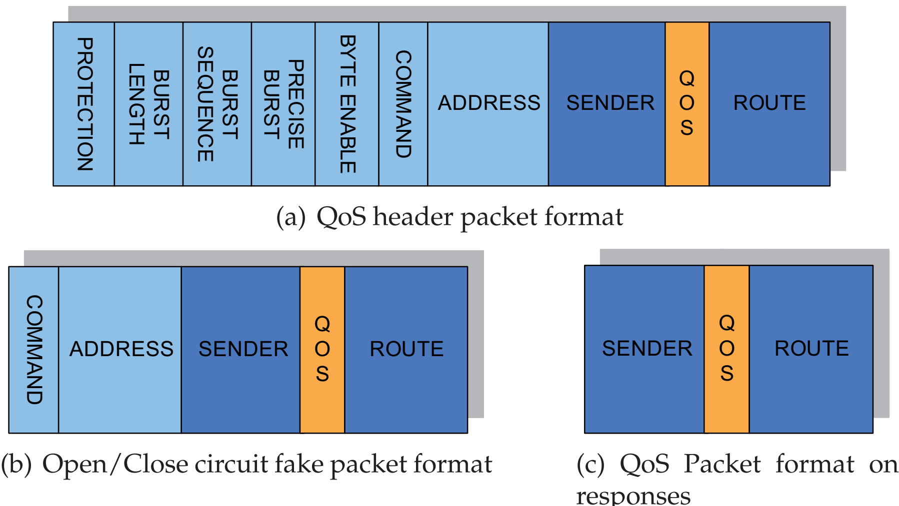
(3) Close the circuit (unidirectional or bidirectional ”fake” transaction).

The packet format for these ”fake” requests and responses transactions are shown in Figure 4.1(b) and Figure 4.1(c) on the following page. In addition, to treat this ”fake” transactions as special, two specific addresses inside the memory-mapped space of each slave have been specified on the NoC-based MPSoC. These special addresses, where ”fake” transactions must point, are placed on MSB bits of the slave memory address space. This is because the MSB byte of the address is used in ×pipes to select a slaves on the NoC-based system. Thus, we choose the next consecutive bits at then end of the slave memory space avoiding the overlapping with the useful address space of the IP core. Next, we have an example, however, this is a configurable parameter that can be fixed as long as is required.

```
#define OPEN_CHANNEL_ADDRESS 0x00E00000
#define CLOSE_CHANNEL_ADDRESS 0x00F00000
```

As shown in Figure 4.1(b) and Figure 4.1(c) on the next page, the packets of the ”fake” transactions have been simplified to minimize the overhead on transmit them. Only the *route*, *source*, *address* and *command* fields together with the *QoS* field are necessary to establish or release a channel between nodes. The minimization is based on the non-necessity to transmit any payload data, but also, it is not necessary to transmit the information related to burst as long as we consider that open/close transactions must be like single request transaction. The *command* filed is used at the initiator NI to know whether the channel is going to be established to perform unidirectional (write transactions) or bidirectional/full duplex (write and read transactions) traffic.

The latency overhead (in clock cycles) to establish or release a channel during the emulation of circuit switching on ×pipes wormhole packet



(a) QoS header packet format

(b) Open/Close circuit fake packet format

(c) QoS Packet format on responses

Figure 4.1: QoS extension on the header packets

switching NoC is determined by the time to setup a request (see Equation 4.1), a response (see Equation 4.2) in conjunction with the number of hops that "fake" packets take to traverse from the source to the destination on the NoC.

$$request_channel = \frac{request_packet_{width}}{FLIT_{width}} + Number_{hops} \tag{4.1}$$

$$response_channel = \frac{response_packet_{width}}{FLIT_{width}} + Number_{hops} \tag{4.2}$$

where $Number_{hops}$ is number of switches that the "fake" packet must traverse on the NoC-based system assuming 1 clock cycle to forward a flit in each switch. On the other hand, $request_packet_{width}$ and $response_packet_{width}$ is the number of bits of the request and response header fields of each type of packet.

$request_packet_{width} = ROUTE_{width} + QOS_{width} + SOURCE_{width} + ADDRESS_{width} + COMMAND_{width}$
$response_packet_{width} = ROUTE_{width} + QOS_{width} + SOURCE_{width}$

The total overhead in clock cycles to open and close a circuit is exactly the same, since we are using the same packet format just changing the QoS bits. However, the overhead is totally different depending whether we require to open/close a circuit in unidirectional (write commands) or in bidirectional or full-duplex mode (write and read commands). A quantification of this

overhead is computed in Equation 4.3 and Equation 4.4, respectively, by using the previous formulas presented in Equations 4.1, 4.2.

$$circuit_uni = 2 \times \frac{request_packet_{witdh}}{FLIT_{width}} \tag{4.3}$$

$$circuit_bid = 2 \cdot \frac{request_packet_{width}}{FLIT_{width}} + 2 \cdot \frac{response_packet_{width}}{FLIT_{width}} \tag{4.4}$$

As shown in Equation 4.1, the overhead under unidirectional traffic is twice the time to transmit request packets in order to open and close the circuit respectively. In bidirectional QoS, is slightly different, on the Equation 4.4 the round trip time (from source to target, and the opposite direction) is defined adding the overhead to send the response packet on the response channel.

Obviously, a part from the modifications at packet level, all FSMs on each NI submodule (i.e. *ni_request*, *ni_rereceive*, *ni_resend*, *ni_response*) have been extended adding a new special state to absorb these "fake" transactions. Within these special states, the transactions are treated as dummy transactions avoiding the generation of non-desirable traffic to/from the target slave.

It is important to notice, however, that the QoS tags must be included on the first flit of the packet header in order to route and grant the packets according the QoS on each switch. As a consequence, in our implementation, the flit width must follow the following rule, $FLIT_{width} \geq ROUTE_{width} + QOS_{width}$.

4.3.2 QoS Extensions at Switch Level

The switch is the key element to lower congestion and improve performance by providing buffering resources, but also is the main hardware component to tune in order to support QoS on the NoC backbone. Its function is to route packets between two endpoint within the NoC.

As shown in Figure 4.2 on the following page, the main blocks of the switch are:

- The *crossbar* provides fully connectivity among the input and output ports.
- The *routing* block is in charge to perform the algorithm to route packets, mainly using XY-YX for Mesh-based NoCs. Although, since ×pipes performs source routing, the switch does not include routing LUTs.

- The *flow control* block implements a credit based (i.e. stall&go) or handshake (i.e. ACK/NACK handshake scheme) flow control protocol in conjunction with control signals from buffers.

- *Input and output buffers* are simply an optimized FIFO or registers to store flits during the forwarding of the packets. However, the size of them is the main trade-off to lower congestion and improve performance on the interconnection.

- The *arbitration* module is attached to each output port to resolve conflicts among packets when they overlap in requesting access to the same output link.

In ×pipes, switches can be fully parameterized in terms of number of input and outputs independently, flit width, and buffer capacity as input and output FIFOs, according the selected flow control protocol.

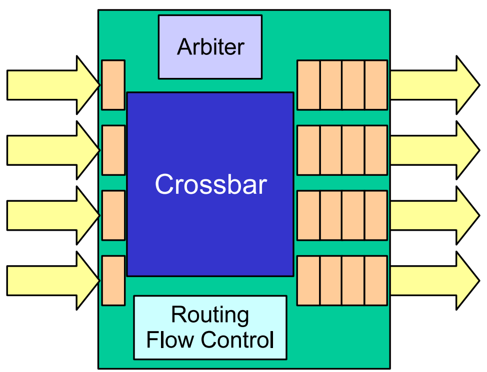


Figure 4.2: Architecture overview of ×pipes switch

From QoS point of view, the main important module is the arbiter, which can be configured on a BE basis for fixed-priority or round-robin policies. In this work, we extend the BE behaviour of the allocator/arbiter block, enabling up to 8-level of priority which is enough to deal with the different use case present in MPSoC, and guaranteeing GT, through the establishment and releasing of exclusive circuits in case it is necessary.

In Figure 4.3(a) on the next page is shown a block diagram of a a general allocator/arbiter of N ports and M levels of priority. The following modification have been done in this module:

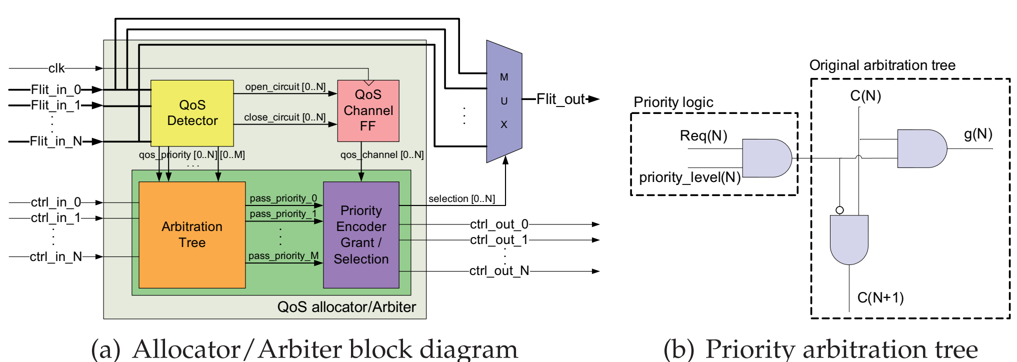
- Include a QoS detector logic.

- Adding a flip-flop register (i.e. `qos_channel`) to store whether the connection is established or not.

- Extending the arbitration tree policy based on fixed priority to enable multiple levels of priority.

- Modify the grant generation using a priority encoder based on the selected priority level, or the `qos_channel` register.

On the allocator (see Figure 4.3(a)), in *QoS detector* is identified which QoS is requested for an specific packet or flow. The QoS field is parsed by selecting the proper range of bits of the first flit of the request header packet. As long as the QoS bits are related to circuit reservation, and depending on its codification (i.e. `ENC_QOS_OPEN_CHANNEL` or `ENC_QOS_CLOSE_CHANNEL`), a set up or tear down of the circuit is done by updating the value on *QoS channel FF* register ('1' if is established and '0' if not). The value of this register is used on the arbitration process to grant the next packet. Once the circuit is established, when the QoS bits are equal to `ENC_QOS_OPEN_CHANNEL`, all the other flows will be blocked until the QoS detector identifies a tear down packet with QoS bits equal to `ENC_QOS_CLOSE_CHANNEL`. Once the circuit is released other flows can progress because the allocator will grant them under BE basis.



(a) Allocator/Arbiter block diagram (b) Priority arbitration tree

Figure 4.3: Allocator block diagram & arbitration tree

On the other side, when multiple packets simultaneously request the same channel on the switch, the allocator grants them using a fixed priority scheme. This rigid behavior have been modified to enable soft-QoS, and whenever the *QoS detector* identify a priority packet (level different from 0) tagged by NI initiator, the allocator will grant it using the priority level embedded on the request header packet (see Figure 4.1(a) on page 92). The arbitration tree reported in [72] has been extended as shown in Figure 4.3(b) for each port of the switch. Depending on the simultaneously requests (inferred using the control lines), the assigned priority levels, in conjunction

with a M level priority encoder, the arbitration tree will produce the `grant` signal.

On both schemes, depending on the `grant` and the `selection` output signals, the switch will choose the output control, and the flit that must go through by using a final multiplexer stage.

However, it is important to remark, that any packet with priority different than 0 has more priority than open/close ”fake” packets, or in other words, GT circuits are created and removed with BE packets that reserve a path through the NoC. Despite this fact, once the circuit is established, the switch will not grant any packet, even if it is very high priority one, until the circuit and the channel is released. Furthermore, the proposed QoS guarantees in-order packet delivery for streaming connections. This is important since packet re-ordering at NI or switch level is impractical on NoCs, first due to the overhead to include additional buffers, and second, because it will increase the packet delivery latency.

4.4 Low-level QoS-support API and Middleware Routines

In order to exploit QoS features in an efficient way, a set of low-level QoS API have been implemented in order to interact with the NIs. These low-level functions interact directly with the NIs and are extremely lightweight (only few assembly instructions) and therefore their execution only takes few clock cycles. In Listing 4.2, we describe the functions of which the low-level QoS API is made up.

Listing 4.2: Low-level middleware NoC services API

```
// Set up an end-to-end circuit
// unidirectional or full duplex (i.e. write or R/W)
inline int ni_open_channel(uint32_t address, bool full_duplex);

// Tear down a circuit
// unidirectional or full duplex (i.e. write or R/W)
inline int ni_close_channel(uint32_t address, bool full_duplex);

// Write a packet with a certain level of priority to a specific address
inline int ni_send_priority_packet(uint32_t address, int data, int level);

// Receive a packet with a certain level of priority from an specific address
inline int ni_recv_priority_packet(uint32_t address, int *data, int level);
```

The purpose of `ni_open_channel()` and `ni_close_channel()` is to open/close the circuits through the ”fake” transactions explained before. This functions receive as parameters, the address of the target, and

a boolean to specify whether the circuit will be established for unidirectional (write) or full duplex (write and read) traffic. The other two routines, `ni_send_priority_packet()` and `ni_recv_priority_packet()` are used to send/receive 32-bit data width according to the priority level to/from an address on the NoC-based system, respectively.

Using a translation table on the NoC-based system, the low-level functions have been encapsulated and extended to manage the priorities and circuits at high levels between any arbitrary source and destination nodes on the NoC-based system. These QoS middleware functions are shown in Listing 4.3.

Listing 4.3: QoS middleware functions

```
// Set high-priority in all W/R packets between an
// arbitrary CPU and a Shared Memory on the system
int setPriority(int PROC_ID, int MEM_ID, int level);

// Reset priorities in all W/R packets between an
// arbitrary CPU and a Shared Memory on the system
int resetPriority(int PROC_ID, int MEM_ID);

// Reset all priorities W/R packet on system
int resetPriorities(void);

// Functions to send a stream of data with GT (unidirectional)
int sendStreamQoS(byte *buffer, int length, int MEM_ID);

// Functions to receive a stream of data with GT (full-duplex)
int recvStreamQoS(byte *buffer, int length, int MEM_ID);
```

Thus, to deal to reset with priorities, `setPriority()`, `resetPriority()` are used to enable assigning a priority level or disable the priority level between two NoC nodes at runtime. An additional middleware routine, `resetPriorities()`, has been included to initialize and restore all flows to non-prioritized traffic removing all priority levels previously assigned.

In `sendStreamQoS()` and `recvStreamQoS()` a circuit is formed from source to destination nodes by means of resource reservation, over which data propagation occurs in a contention-free regime. In `sendStreamQoS()` because the traffic is only from source to target, the circuit is open in unidirectional, while in `recvStreamQoS()` the reservation of the channel is done in both directions allowing GT on the reception of the stream.

For further information about the use of this functions refer Section 6.8 on page 166, where they are used to apply QoS on OpenMP parallel regions boosting and delaying threads on the parallel programming model.

4.5 QoS Verfication and Traffic Analysis

In this section we perform functional verification and simulation under different traffic using the QoS features explained before. The aim is to ascertain the correct behaviour under different scenarios of the QoS features at NoC level, as well as to verify the interaction of low-level and middleware functions when complex transactions patterns are generated from software applications.

We use MPARM [54], a cycle-accurate modeling infrastructure to investigate the system-level architecture of MPSoC platforms. MPARM is a plug-and-play platform based upon the SystemC [45,46] simulation engine, where multiple IP cores and interconnects can be freely mixed and composed. At its core, MPARM is a collection of component models, comprising processors, interconnects, memories and dedicated devices like DMA engines.

The user can deploy different system configuration parameters by means of command line switches, which allows for easy scripting of sets of simulation runs. A thorough set of statistics, traces and waveforms can be collected to analyze performance bottlenecks and to debug functional issues.

In terms of interconnect, MPARM provides a wide choice, spanning across multiple topologies (shared buses, bridged configurations, partial or full crossbars, NoCs) and both industry-level fabrics (AMBA AHB/AXI [14, 32], STBus [28]) and academic research architectures (×pipes [52,53,69]). On top of the hardware platform, MPARM provides a port of the uClinux [143] and RTEMS [182] operating systems, several benchmarks from domains such as telecommunications and multimedia, and libraries for synchronization and message passing.

Using MPARM, we developed a SystemC cycle accurate model of 4-core NoC-based MPSoC architecture using the custom topology presented in Figure 4.4 on the next page. The NoC-based system have been configured using ×pipes compiler as follows:

- Three switches of 5x5 ports.
- Four ARM processors with its private memories placed from 0x00-0x04.
- A shared memory, semaphore and an interrupt device placed at 0x19, 0x20 and 0x21 on the system address space, respectively.
- All QoS features developed in this chapter (QoS NIs and extended switches).

- The NoC uses wormhole packet switching, stall&go flow control, and fixed priority arbitration if not priority packet is requested.

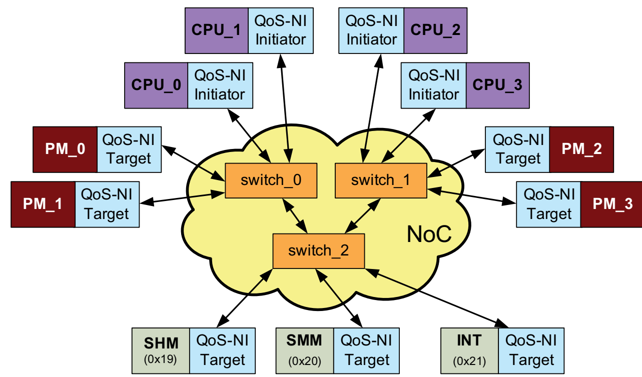


Figure 4.4: Topology 5x5 clusters NoC-based MPSoC

In order to validate our QoS support in the previous architecture, we develop a full function verification framework to demonstrated the strength of the designed QoS. From each processor, we inject continuously traffic towards the shared memory placed at address 0x19, which automatically will generate congestion on the NoC backbone. Four different scenarios have been tested:

- **Scenario 0:** No priorization traffic.
- **Scenario 1:** Only `CPU_0` traffic is not prioritized.
- **Scenario 2:** the right cluster (i.e. `CPU_2`, `CPU_3`) traffic is prioritized, over the cluster on the left (i.e. `CPU_0`, `CPU_1`).
- **Scenario 3:** Only `CPU_3` traffic is prioritized.

The results are shown in Figure 4.5 on the following page. On scenario 0, the order of completion of the CPUs is: 0, 1, and then 2, 3. This is because, no traffic priorization have been defined using the middleware routines, and therefore, the fixed priorities defined in the switches (see Equation 4.5) are resolving the concurrent access to the shared resource.

$$CPU_{P0} > CPU_{P1} \;\; and \;\; CPU_{P2} > CPU_{P3} \tag{4.5}$$

and the cluster on the left, has more priority than the right cluster, on the `switch_2`.

On scenario 2, when all processors are prioritized except CPU_0, we can see the influence of changing at runtime the priorities because all processors finish before CPU_0. On scenario 3, we simulated a similar approach, reversing the fixed priorities, so that the cluster on the right has more priority than the one on the left. As expected, CPU_2, CPU_3 finish before CPU_0, CPU_1, in contrast to scenario 0. On scenario 3, we assessed the effect to prioritize only CPU_3. As we can observe, CPU_3 finishes first, almost at the same time that CPU_0 and CPU_1, but the collateral effect, is that CPU_2 is delayed since it is always less prioritized when it competes with the rest of processors.

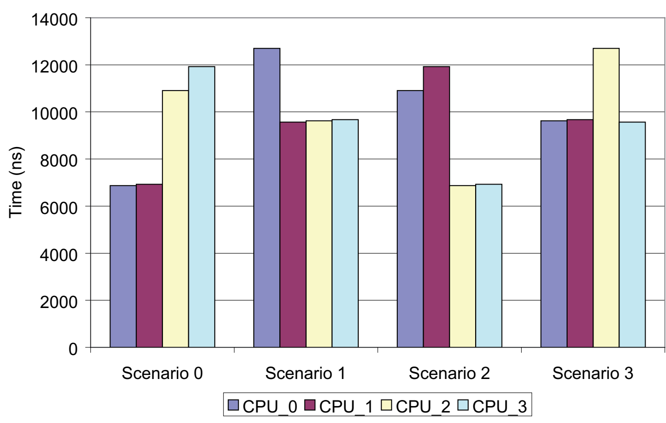


Figure 4.5: Validation and traffic priorization under different scenarios

4.6 Synthesis results

In this section, a quantification of the overhead, in terms of resource usage, and degradation of the circuit performance, in terms of f_{max} of the implemented QoS features is reported by doing a synthesis of the components on different FPGAs devices[2].

A detailed quantification have been done at two levels: *(i)* at NI level, *(ii)* at switch level, comparing the circuit blocks with and without the QoS extensions. The components have been fixed to 32 bits for flit width, as well as addresses and data buses, while the switches have been configured under a minimal buffer capabilities (2 flits input buffers).

[2]The results have been extracted using Synplify® Premier 9.4 [167] to synthesize each component on different FPGAs without any optimization. VirtexII (xc2v1000bg575-6) and Virtex4 (xc4vfx140ff1517-11) from Xilinx, and StratixII (EP2S180F1020C4) and StratixIII (EP3SE110F1152C2) from Altera.

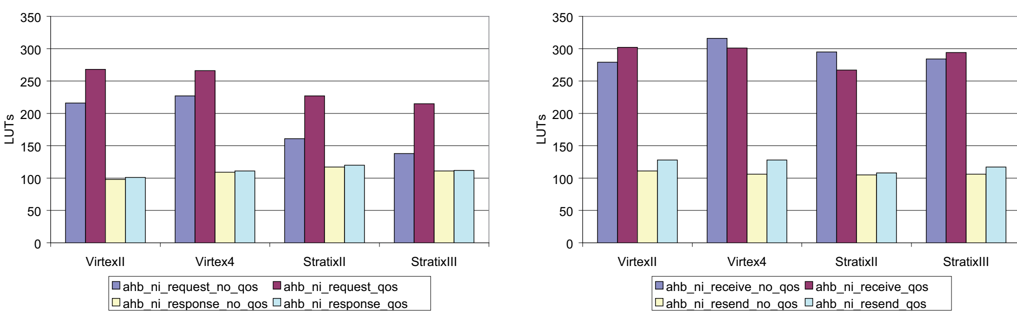
Note that all synthesis reported in the next sections are before place-and-route without any optimization. After layout, normally the area increases and the maximum frequency drops.

4.6.1 QoS Overhead at Network Interface

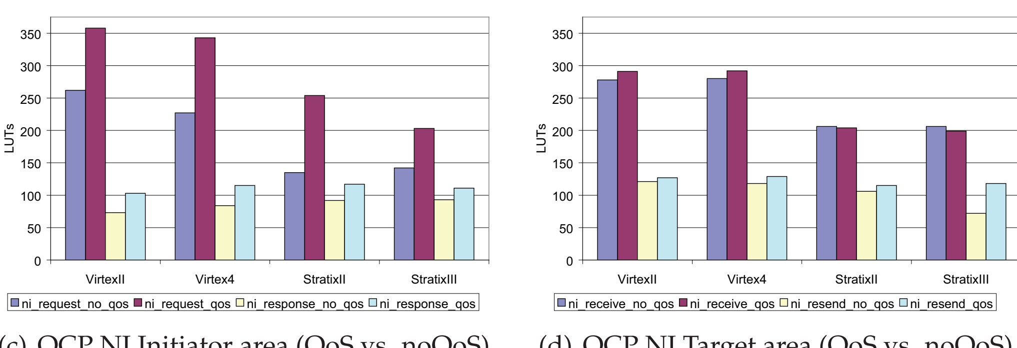
The QoS features have been implemented on the AMBA 2.0 AHB network interfaces presented in Chapter 3 on page 57 but also on the OCP network interfaces already present in ×pipes [52, 53].

At first glance in Figure 4.6, it is easy to observe that, the major overhead at NI level is present on *ni_request* module. This result is easy to understand, since we include on it the mechanism to trigger QoS packets (a QoS encoder, which codify the bits of QoS field) using a set of configuration registers, as well as several changes on the request FSM to generate the "fake" transactions.

On the remaining modules, such as *ni_receive*, *ni_resend*, *ni_response*, the QoS extensions do not impact significantly, since only small changes have been done on the FSMs of each module.



(a) AMBA NI Initiator area (QoS vs. noQoS) (b) AMBA NI Target area (QoS vs. noQoS)



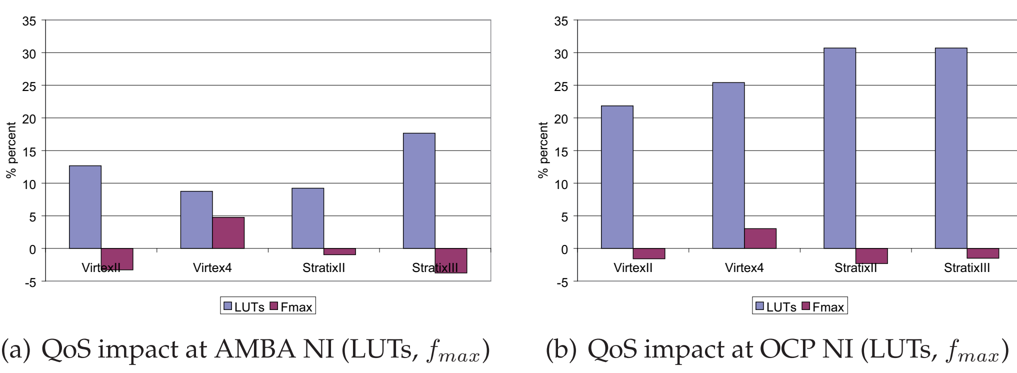
(c) OCP NI Initiator area (QoS vs. noQoS) (d) OCP NI Target area (QoS vs. noQoS)

Figure 4.6: AMBA – OCP Network Interfaces (QoS vs. noQoS)

However, even if the trend on OCP NIs (see Figure 4.6(c) and Figure 4.6(d) on the previous page) is similar as in the AMBA 2.0 NIs (see Figure 4.6(a) and Figure 4.6(b) on the preceding page), the overhead is a little bit more significant on the OCP NIs, specially on *ni_request*.

A summary in each FPGA of the effect to include QoS extensions for AMBA 2.0 AHB and OCP NIs are presented in Figure 4.7(a) and in Figure 4.7(b), respectively. Thus, computing the average, we can affirm that the extended AHB NIs with QoS has around 12% overhead area costs, whereas for OCP NIs this overhead increase until 27%. However, the area resources overheads can be assumed since the percentages are related to NoC components of hundred LUTs, and therefore, at the end of the day, we are talking about less than tens LUTs of overhead.

In contrast, as shown in Figure 4.7, the f_{max} degradation of the circuitry of both NIs can be strictly considered negligible since the average among the FPGA devices is around 0%. Surprisingly in some cases, for instance in Virtex4 FPGAs, the synthesis give better f_{max} for the extended NI with QoS than without, even if a priori, the QoS NI is a little bit more complex. In addition, normally, the critical paths are on switches, and therefore, the f_{max} at NI level is not specially very critical.



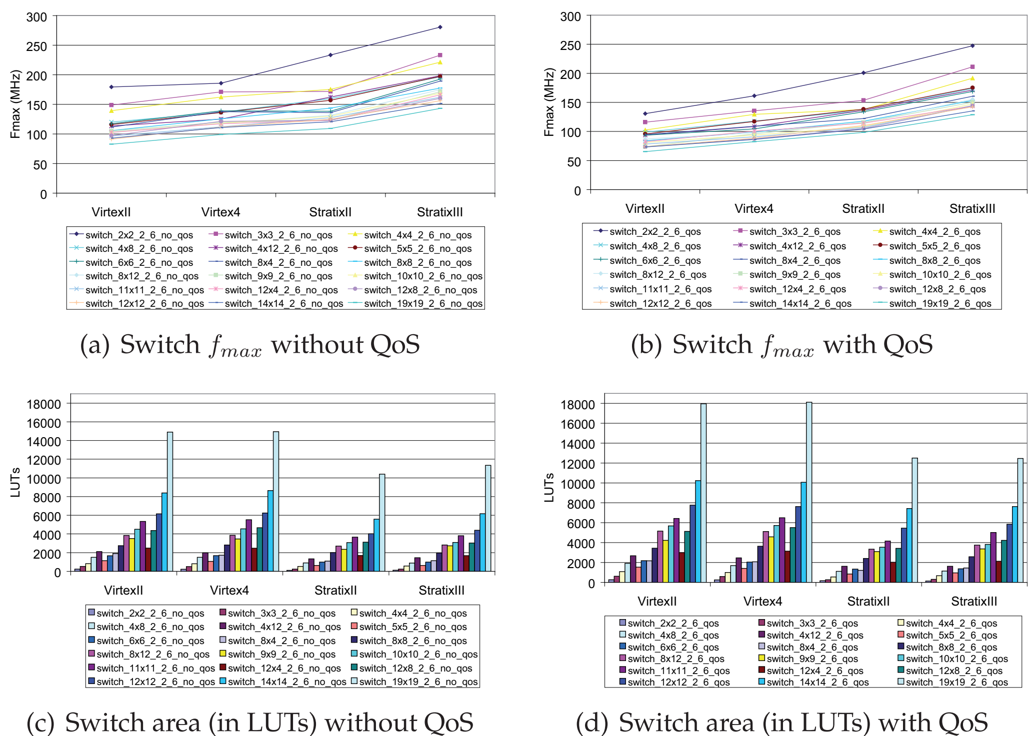
(a) QoS impact at AMBA NI (LUTs, f_{max}) (b) QoS impact at OCP NI (LUTs, f_{max})

Figure 4.7: QoS impact (LUTs, f_{max}) at NI level (AMBA-OCP)

4.6.2 QoS Overhead at Switch Level

In this section, we focus on explore the synthesis results carried out on multiple switches by increasing the cardinality in order to evaluate the QoS overhead in different FPGA devices. Because the implemented QoS features is strongly dependent on the port number, as commented before, we configured the switches with multiple arities, fixing the flit width 32-bits.

In Figure 4.8 we show the synthesis results (LUTs, f_{max}) of multiple switch configurations with and without the QoS extensions[3]. As expected, without QoS, the behaviour in all figures reflects that when the switch radix increase the area increases, and the frequency drops. The same behaviour can be extracted from the plots showing switches with QoS extensions. However, when QoS is included, it is easy to observe that, the increment is exponentially, and the frequency drops, but not dramatically.



(a) Switch f_{max} without QoS (b) Switch f_{max} with QoS

(c) Switch area (in LUTs) without QoS (d) Switch area (in LUTs) with QoS

Figure 4.8: Switch synthesis varying the arity (QoS vs. noQoS)

A summary of the results computing the average area overhead and the decrease of the operating frequency among each switch configuration in each FPGA device is reported in Figure 4.9 on the following page. As shown in Figure 4.8(a), the f_{max} range on the fastest FPGA is comprised from 280,6 MHz and 143,2 MHz, among the smallest and largest switch configuration without QoS, on StratixIII devices. Including the QoS extensions, on the same FPGA, the f_{max} range drops to 247,5 and 128,8 MHz. As shown in Figure 4.9 on the following page, the operating frequency of the switch with QoS drops between 11-20% in each FPGA device.

[3]QoS extension have been configured to support circuit reservation and only two levels of priority.

Similarly, comparing the area resources among switches with and without QoS, as shown in Figure 4.9, the overhead is quantified between 23-28% depending on the FPGA device.

Definitely, even if the area overhead and the frequency drop on the switches can not be considered negligible, but the QoS extension certainly can be considered a low-cost implementation, for its simplicity, avoiding the use of virtual channels and complex table-based TDMA schemes to guarantee latency and bandwidth, which increments the area resources on each switch. Instead, a unified buffer priority scheme and the emulation of a simple but well-known circuit switching on a wormhole packet switching network have been implemented successfully.

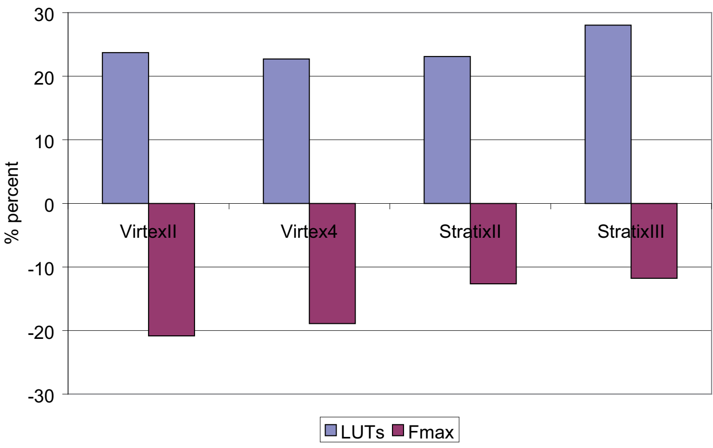


Figure 4.9: QoS impact (f_{max}, LUTs) at switch level

However, next in Figure 4.10 on the facing page we explore more in deep, the scalability varying the number of priority levels on each of the switch configurations presented in Figure 4.8 on the previous page. As expected, when 8 priority levels are used, the area costs increments considerably, concretely, the area resources are incremented close to 100%, i.e. doubling the area of the switch without QoS. Despite this fact, the area resources are moderately quite closer when we 2 or 4 priority levels are use, ranging from 23-40% in Virtex FPGAs. In Stratix the trend is similar, but now the overhead and the range to include 2 or 4 levels is more wide going from 25-56% for the worst case.

On the other side, in terms of f_{max}, as shown Figure 4.10 on the facing page, the frequency drops fast when more levels of priority are included on the switch. Despite this fact, the curve tends to stabilize when more than 8 levels are implemented. In fact, the possible scenarios, traffic classes or use

cases, on the embedded domain can fitted using up to 8 levels of priority.

Thus, a priori, unless it is a very specific application, it will be not necessary to extend the switch with more than eight levels of priority. In fact, it is easy to observe that, the overhead to include 2 and 4 priority levels, even if it is not negligible, can be assumed taking into account the potential benefits to have runtime QoS on the final system.

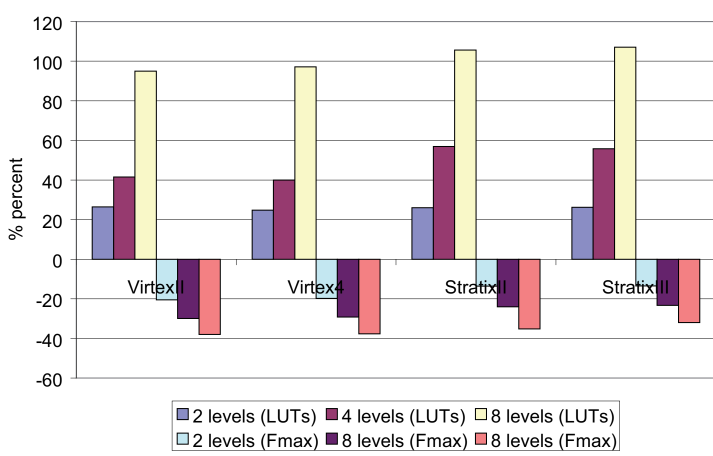


Figure 4.10: QoS impact (LUTs, f_{max}) according to the priority levels

4.7 Conclusion and Future Directions

NoC-based systems relies on a OSI-like protocol stack (see Section 2.1.7 on page 31) which lets to include NoC services supported by hardware at NoC level within the switch. In addition, our approach, is based to control the QoS features from software layers (by transport layer at network interfaces) until the application level and on the parallel programming model (see Section 6.8 on page 166).

Predictability on homogeneous, but specially on heterogeneous parallel NoC-based MPSoC needs to be addressed by providing a right mix of soft and hard real-time guarantees.

Moreover, thanks to predictability, individual IP cores with a great variety of traffic requirements can be more easily integrated on generic NoC-based MPSoC. For instance, a multimedia real-time cores can be integrated effortlessly with low rate UART I/O cores, facilitating the analytical verification models.

In this work, we introduced a complete framework to support QoS-related features until the software layers. We develop a reasonable low-cost QoS combining BE an GT at NI and switch level on the NoC backbone. Additionally, a set of low-level and middleware functions are provided to reconfigure the traffic behaviour on the NoC backbone whenever it is required. Based on both, the low-cost hardware design, and a lightweight software implementation of the middleware API, we can achieve an extremely efficient runtime QoS reconfiguration support on our NoC-based MPSoC.

This hardware-software infrastructure can be exploited effectively as well by QoS managers or OS scheduler to steer NoC fabric, ensuring end-to-end QoS, and reconfiguring it, in case of congestion.

Short term future work, is to include QoS on top of parallel programming models. In fact, in Section 6.8 on page 166, the QoS extension at NoC level have been integrated by means of QoS middleware API on a runtime OpenMP library presented in [183]. Clearly, the next step in the same direction will be to extend our message passing communication library, ocMPI (see Section 6.5 on page 146) to provide GT during message passing.

Future directions are to provide the rest of NoC services at runtime, such as performance monitoring and tuning, DVFS and dynamic power management, etc. As in this work, all these services must be made available through hardware components and software libraries to reconfigure the chip as long as is necessary based on software directives.

CHAPTER 5

HW-SW FPU Accelerator Design on a Cortex-M1 Cluster-on-Chip

This chapter[1] will face a HW-SW design of a Floating Point Unit (FPU) on a Cortex-M1 MPSoC cluster. A full-featured design at hardware level of a FPU as a decoupled accelerator have been done in conjunction with a software API to perform floating point operations in a transparent way from the user point of view (i.e. without doing any changes on the ARM compiler).

5.1 Motivation and Challenges

To face the incremental design complexity in next generation NoC-based MPSoCs, the efficient integration and reuse of IP cores is essential to integrate high complex functionality on a single chip. However, this approach imposes significant challenges, mainly on the field of the communication between hardware and software components on the system.

Nowadays, most MPSoCs due to the inherent requirement to compute complex software applications and kernels, make use of floating-point arithmetic, whether for occasional calculations, or for intensive computation kernels. The most relevant applications in the area of embedded systems are:

- Digital multimedia processing of high-quality audio and video.
- Digital signal processing tasks, particularly spectral methods such as DCT/iDCT, FFT.

[1]The author will like to thanks contributions by Per Strid on Cortex-M1 Polygonos MP-SoC platform, and John Penton on the development of Cortex-R4F processor, as well as the rest of brilliant researchers that work at ARM R&D Department in Cambridge

- 3D Gaming and graphic processing (computation of floating point data sets within rotation, transformation, etc).
- Matrix inversion, QR decomposition, etc, in software defined radio or wireless communications and radar kernels.
- In sophisticated control algorithms on embedded automotive applications.
- Computation intensive applications in real-time systems.

Often, in low-cost and non-critical performance systems, all these floating point operations are emulated and executed using uniquely a floating point software library without any hardware support. However, to achieve high performance embedded computing, in next generation of embedded MPSoCs architectures exist the clear requirement to implement them in hardware rather than in software.

The *IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)* [184] is the most widely-used standard (since 1985) for floating-point computation, and is followed by many CPUs, compilers and custom hardware FPU implementations. The standard defines formats (see Figure 5.1 on the next page) for representing floating-point numbers in single and double precision (i.e. including zero and denormal numbers, infinities and NaNs) and special values together (such as $\pm\infty$ or ± 0 values as shown in Figure 5.1(a) on the facing page). The IEEE 754 representation is composed by three main fields, *(i)* sign, *(ii)* exponent and *(iii)* mantissa. As shown in Figure 5.1(a) and Figure 5.1(b) on the next page, depending on the precision the width of these fields are slightly different. A set of floating-point operations (e.g. add, subtract, multiply, division, square root, fused-multiply-add, etc) have been defined robustly on top of this representation, as well as type conversions (int to float, float to double, etc), and manipulation of the sign on these values (e.g. abs, negate). The standard also specifies four rounding modes when a floating point operations is performed:

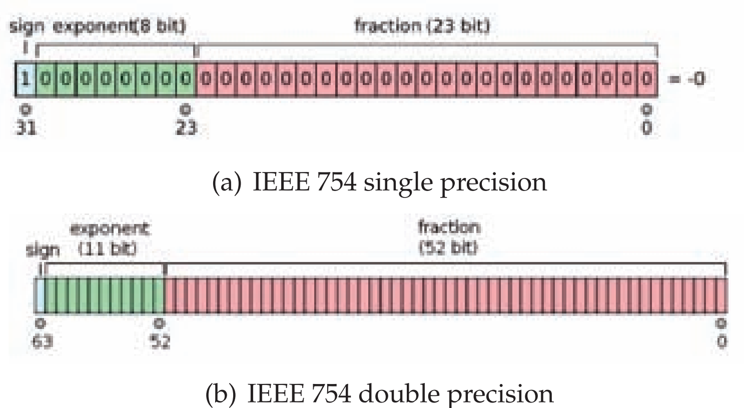
(1) *Rounding to nearest (RN):* rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers).

(2) *Rounding towards +inf (RP)*: directed rounding towards $+\infty$.

(3) *Rounding towards -inf (RM)*: directed rounding towards $-\infty$.

(4) *Rounding towards zero (RZ)*: directed rounding towards 0 (also called as truncation).

and five exceptions, such as, *invalid operation* (e.g. square root of a negative number), *divided by zero*, *overflow/underflow* (if a result is too large or too short to be represented correctly) and *inexact*.



(a) IEEE 754 single precision

(b) IEEE 754 double precision

Figure 5.1: Single and double precision IEEE 754 representation

In this chapter, we will focus on the exploration of different alternatives to integrate a floating point unit taking into account all hardware and software issues.

We will first outline how we design in hardware an AMBA-based standalone FPU reusing an existing one targeted to accelerate floating point operation for the Cortex-R4F processor [185]. We will then explain how to integrate the FPU in a full-featured MPSoC cluster based on ARM Cortex-M1 [186], the ARM soft-core targeted to FPGA devices. Finally, we will explore the different methods to communicate the designed FPU with the processor, under different scenarios, based on a crossbar or NoC backbones.

5.2 Related Work

Nowadays, FPGA platforms are emerging as HPC systems, in terms of GFLOPS/Watts, implementing parallel double floating points coprocessors to enable the concept of Green Computing [187,188] thanks to great amount of embedded DSP blocks (i.e. adders, multipliers, etc.) integrated in last FPGA devices.

In addition, main FPGA vendors such Altera or Xilinx offer the possibility to include custom hardware FPUs together with their soft-core processors. Altera offers the FPU as a custom instruction inside the Nios II processor [20], and also the possibility to create standalone Altera Megafunction [189] (e.g. addition/subtraction, multiplication, division, logarithm,

etc.) according with the floating point operation that has to be accelerated. Xilinx provides an equivalent approach, the user can optionally integrate a FPU on MicroBlaze or PowerPC [190] processors, or a custom floating point operation [191]. Thanks to the use of soft-core processors, such as Nios II [20] and MicroBlaze [18], and depending on the requirement of our system, we can choose whether to integrate the FPU or not.

Other relevant FPU implementation is GRFPU, an AMBA AHB FPU compliant with SPARC V8 standard (IEEE-1754) provided in conjunction with LEON3 processor [16].

Generally, these FPUs can be integrated as a tight-coupled coprocessors with each CPU to compute floating point operations purely in hardware, and all of them are IEEE 754 compliant. Normally, they support single or double precision addition, subtraction, multiplication, division, square root, comparison, absolute value, etc, and sign change operations, as well as built-in conversion instructions to transform from integer to floating point types and vice versa.

Xtensa processors also provide these capabilities extending the base ISA by using Tensilica Instruction Extension (TIE) [192,193] to add new features as for instance AES/DES encryption, FFT and FIR operations, double precision floating point instructions. Thus, the processor can be tailored to a particular set of SoC applications delivering high-performance and energy efficiency at reasonable cost, becoming an ASIP-like processor.

In this work, we face a similar approach to the presented before to accelerate the existing software emulation libraries using IEEE 754 floating point representation. Thus, it is immediately apparent that there is value in re-designing and reusing an existing FPU from ARM, concretely ARM Cortex-R4 FPU, to be the FPU of Cortex-M1 processor, but also in a Cortex-M1 MPSoC cluster on chip. In addition, from research point of view, there are many important issues to explore:

- Hardware and software communication interfaces.
- Study the viability to share the FPU among several processor by exploring different synchronization methods.
- Test the performance of the hardware-software communication and synchronization protocols.
- Atomic memory accesses on different interconnection fabrics.
- Cycle count comparison on the execution of different floating point instructions.

- Speedup between our FPU accelerator floating point functions versus the existing software emulation ARM floating point library.
- Evaluate the increment in area when the HW floating point support is included in the system.

In addition, since this dissertation is mainly focused on MPSoCs and the evaluation of the interconnection fabrics, we provide analyses reporting results under two alternative fabric architectures, i.e. an AMBA AHB ML crossbar fabric [27], and on a ×pipes NoC backbone.

We believe that the implementation and experimentation of such a platforms is crucial in the context of the main goal of this dissertation, i.e. the assessment of the tradeoffs involved in the communication of hardware and software components on MPSoCs, as well as, providing hardware support during the execution of a floating points operations to achieve high performance parallel embedded computing.

5.3 Cortex-M1 FPU Accelerator – HW Design

In the previous section, we listed the most important FPUs together with the corresponding CPU mainly targeted to FPGA devices or to ASIC technologies, specially in case of GRFPU-LEON3, and the Xtensa TIE processors. Next, we describe our proprietary design of a custom FPU accelerator based on the existing on Cortex-R4F FPU. Our FPU accelerator will be targeted to be included in conjunction with Cortex-M1 soft-core processor. However, to face the hardware design of the FPU, two restrictions have been imposed in order to preserve Cortex-M1 architecture:

- Must be done without doing any changes on the Cortex-M1 core, and by extension to the ARM-v6M architecture.
- Must be completely independent from the ARM compiler.

Both restrictions imply that the FPU can not be tight-coupled on Cortex-M1 architecture, inside the processor datapath, as it is done in Cortex-R4F [185] CPU, and in the rest of the related work. In this work, the FPU accelerator will be designed to be standalone and attached to an AMBA 2.0 AHB interconnection fabric (i.e. bus, crossbar or NoC backbone).

The Cortex-R4F processor's FPU performs single and double floating-point calculations that allow a greater dynamic range and accuracy than fixed-point calculations. The FPU is IEEE 754 compatible and is backward compatible with earlier ARM FPUs (VFP9/10/11). The implementation is

optimized for the single precision (SP) processing without sacrificing double precision (DP) support. The architecture is presented in Figure 5.2. Basically, the FPU architecture is embedded in the Cortex-R4 processor datapath, using multiple stages of pipeline (fetch, decode, execution, and write back), and it has also its own 32-bit register file. As it is tight-coupled with the CPU, it makes use of the CPU ALU to execute integer external multipliers necessary to perform floating point operations. The architecture is divided in two parts: *(i)* a dual pipeline architecture for SP, and *(ii)* and unified compact pipeline for DP flating point operations. Since in this work we will support SP operations, we will focus on analyze the dual SP pipeline architecture.

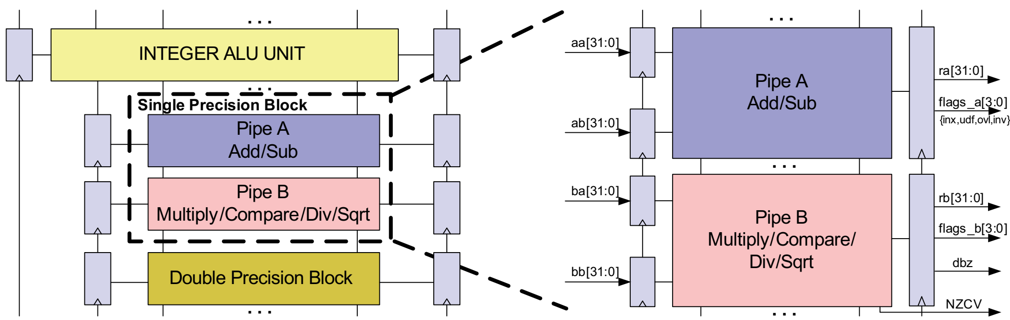


Figure 5.2: Cortex-R4 Floating Point Unit (FPU)

As shown in Figure 5.2, the idea is to cut the single precision (SP) block of this complex tight-coupled FPU to be reused as the back-end of our SP FPU accelerator. In this work, we mainly focus on the front-end design, together with the integration and validation of the existing back-end, in order to implement a SP FPU compatible with AMBA 2.0 AHB. The new design, rather than a tight-coupled FPU, will be an AMBA-based memory mapped standalone FPU accelerator for the Cortex-M1 (see Figure 5.3 on the facing page). The front-end implementation includes the following features:

- An AMBA 2.0 AHB front-end interface, fully compatible with AMBA 2.0 AHB [14] or AHB ML standards [27].
- A simple memory register bank.
- An AMBA-based control unit.
- A semaphore to block/unlock exclusive access to the FPU.
- A micro-instruction decoder.

- The external multipliers required.

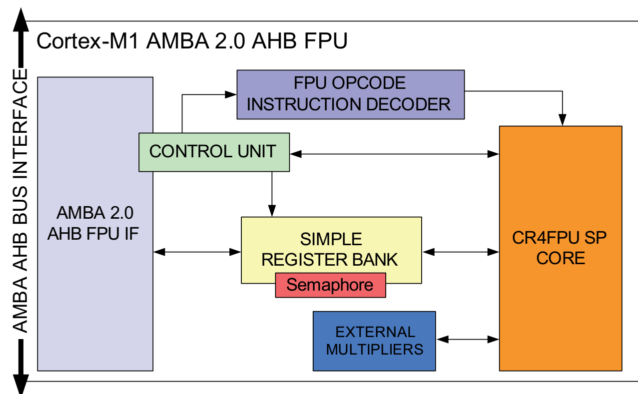


Figure 5.3: Cortex-M1 AHB SP memory mapped FPU accelerator

The AMBA AHB interface module is in charge to translate control and data AHB transactions from the CPU towards our FPU. Since the FPU will be attached upon AMBA-based systems, a FSM will be designed as a control unit of the FPU. This control unit will supervise the CPU-FPU protocol presented in Section 5.4 on page 116, by asserting/de-asserting properly the control lines to/from the other blocks in our SP FPU accelerator. To support the same features as the tight-coupled ARM FPU, in our standalone design an equivalent register bank have been included. The register bank of the FPU is presented in Figure 5.4. It consists on multiple R/W registers memory mapped on the address space of the designed SP FPU accelerator. This front-end has the same features as its tight-coupled version, and it is extremely slim and latency efficient, since it only takes one cycle delay for writes, and two cycles for read operations on the register bank.

Register	Address
FPSID (32-bit RO)	AddressBase + 0x0
FPSCR (32-bit RW)	AddressBase + 0x4
FPEX (32-bit RW)	AddressBase + 0x8
FP Operand A (32-bit WO)	AddressBase + 0xC
FP Operand B (32-bit (WO)	AddressBase + 0x10
FP Opcode (32-bit WO)	AddressBase + 0x14
Result PipeA (32-bit RO)	AddressBase + 0x18
Result PipeB (32-bit RO)	AddressBase + 0x1C

Figure 5.4: SP FPU accelerator memory mapped register bank

In our register bank within our AMBA-based SP FPU accelerator, we can distinct logically two different types of registers:

- 32-bit registers to use as storage of floating point numbers.
- FPU system registers.

Basically, the registers to use with floating point operations are *FP Operand A*, *FP Operand B*, used as write-only inputs operands, and two read-only output result registers connected to each pipe, i.e. *Result Pipe A*, *Result Pipe B*, which save the final result of the selected operation in *FP Opcode* register. The *FP Opcode* register, shown in Figure 5.5, uses 9 bits to save the opcode (i.e. FADDS, FSUBS, FMULS, FTOSIS, FDIVS, FSQRTS, etc.) during its execution, and furthermore, it is used to configure two different flags (i.e. default NaN or flush to zero), as well as one out of four rounding modes specified in IEEE 754 floating point standard.

FP_OPCODE[31:13]	FP_OPCODE[12]	FP_OPCODE[11]	FP_OPCODE[10:9]	FP_OPCODE[8:0]
Reserved	DEFAULT_NAN	FLUSH_TO_ZERO	ROUND_MODE	OPCODE_INSTRUCTION

Figure 5.5: Floating Point OpCode register (FP Opcode) layout

The system registers (i.e. *FPSID*, *FPSCR*, *FPEX*) are used to configure and examine the status of our SP FPU accelerator.

- *Floating Point System ID register (FPSID):* read-only 32-bit register which stores ARM specific information (see Figure 5.6), such as the variant and revision, if it is tight-coupled coprocessor or it use a SW library, and whether it supports single and double precision (D variant) or not.

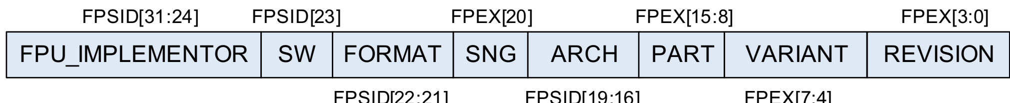


Figure 5.6: Floating Point System ID register (FPSID) layout

- *Floating Point Status and Control register (FPSCR):* configuration and status control register used to read/write 32-bit register used to check *NZCV* conditions (i.e. less than, equal, greater than or unordered result) flags used in floating point comparisons, but also to examine whether an exception occurs during the execution of a floating point operation.

FPSCR[31:10]	FPSCR[9:6]	FPSCR[5]	FPSCR[4]	FPSCR[3]	FPSCR[2]	FPSCR[1]	FPSCR[0]
Reserved	NZVC	IFZ	INEXACT	UNDERFLOW	OVERFLOW	DIVISION BY ZERO	INVALID OP

Figure 5.7: Floating Point Status and Control register (FPSCR) layout

- *Floating Point Exception Register (FPEX):* read/write 32-bit register used to deal with code exceptions through EX, and EN bits. Inside this register, on the reserved bits for user defined implementation, we map two special features of our SP FPU accelerator. We include an embedded semaphore bit, i.e. *FPU_BUSY* on FPEX[1], to enable the capacity to share the FPU in a MPSoC among different processors as we shown in Section 5.6 on page 122, and another bit to check from the CPU whether the floating point operation is finished or not, i.e. *DATA_READY* on FPEX[0].

FPEX[31]	FPEX[30]	FPEX[29:2]	FPEX[1]	FPEX[0]
EXbit	ENbit	Reserved	FPU_BUSY	DATA_READY

Figure 5.8: Floating Point Exception Register (FPEX) layout

On the other side, the back-end integrated in our SP FPU accelerator is almost the same as used in Cortex-R4F, with a set of integer embedded multipliers required for the core. Thus, as shown in Figure 5.2 on page 112, the wrapped SP has been divided in two pipes, *Pipe A* and *Pipe B*. *Pipe A* is for addition/subtraction and, *Pipe B* is used for multiplication, division, square-root, as well as the comparisons and conversions among integer and standard float C/C++ data types.

On pipe B, the multiplication is pipelined, but the division and the square-root use an iterative algorithm stalling the pipeline of the FPU whenever is required. This issue cause the incompatibility to execute pipelined multiplication and divisions. Thus, in our design, each floating point operation is not pipelined due to the fact that is connected through an interconnection fabric rather in the CPU datapath.

On both pipelines, results are stored in the WR stage on a 32-bit register. Moreover, the FPU also stores, in FPSCR register, the exceptions and flags generating the corresponding value according to the operation executed on the back-end.

This dual-pipe architecture of the back-end, and the slim and efficient front-end will deliver the best performance area trade-offs when Cortex-M1 executes floating point operations using our SP FPU accelerator.

5.4 HW-SW CPU-FPU Communication Protocol

As explained before, normally the FPU is embedded in the processor datapath as a tight-coupled coprocessor. However, in our approach the SP FPU accelerator is completely standalone and memory mapped on the system. This architecture opens different alternatives when the CPU needs to issue a floating point instruction and communicate to the external FPU accelerator.

In Figure 5.9 on the next page, our CPU-FPU protocol at transaction level is presented on an AHB-based system, whereas in Listing 5.1, we enumerate the main points, consisting in six basic transactions.

Listing 5.1: CPU – FPU protocol definition

```
Step 0 - Check if the FPU is not busy (only if it is shared)
Step 1 - Transfer OperandA
Step 2 - Transfer OperandB
Step 3 - Transfer OpCode
Step 4 - Wait until the result is ready
Step 5 - Read flags (optional or by IRQs)
Step 6 - Read result
```

Principally, this CPU-FPU protocol can be viewed in three main group of transfers:

(i) Check whether the FPU is busy or not computing the previous floating point operation (Step 0).

(ii) FPU configuration (Steps 1-3), based on three 32-bit write transactions involving the operands and the opcode of the floating point unit.

(iii) Wait until the result is computed, and afterwards collect it (Steps 4-6).

In *(i)*, the principle is to enable the ability to share the FPU among several processors in the system. This means that at any time, any processor or specific piece of hardware can issue a floating point instruction to our FPU, and therefore, some synchronization is required. Furthermore, since the proposed protocol in Listing 5.1 performs individual transactions to access and configure the FPU, these transactions must be atomic in order to avoid other CPU access the FPU meanwhile is computing. Normally, atomic memory accesses are implemented by locking AHB bus. As long as one master locks the bus, the arbiter does not grant it to any other master. Unfortunately, the architecture ARM-v6M and by extension the Cortex-M1 does not support exclusive access transactions which blocks the interconnection fabric.

In our design, synchronization and atomicity have been provided using a traditionally lock-based method to support concurrent access to the shared resource. This process restricts the independence of processes by enforcing a certain order of execution between the consumer and the producer. The synchronization is implemented using a semaphore embedded in the FPEX register. In the simplest synchronization case, one or more processors competing for our shared FPU resource may poll the semaphore to gain resource access. Once, a CPU gets access to the FPU, it automatically asserts the *FPU_BUSY* bit (see in Figure 5.9 how FPEX value change to 0x40000002) to block other processors which potentially could try to require access to that hared resource. When the floating point computation finish, and after, the result has been read by the requesting CPU, the FPU unlocks by de-asserting *FPU_BUSY* bit. This process will ensure at the same time the atomicity of the protocol enabling the shareability in a multi-core system.

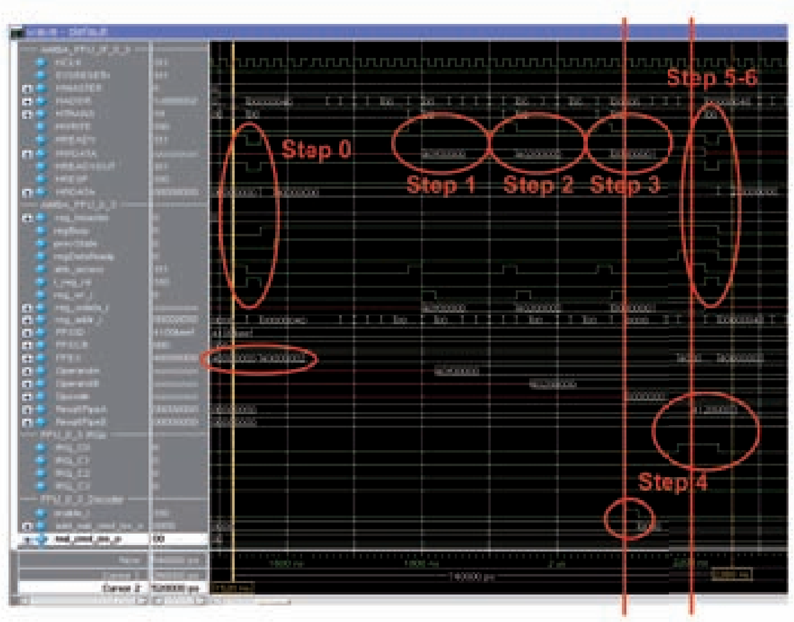


Figure 5.9: CPU – FPU hardware-software protocol

On a NoC-based MPSoC, the synchronization and the memory access atomicity can be done by blocking the NoC channel (inside each allocation on the switch involved in the path) using the QoS features presented in Section 4.3 on page 89. This QoS support enables exclusive transmission/reception of memory transactions, and consequently, the granted master can

perform atomic memory operations since no other processor will have access to the FPU. Despite locked transfers are sometimes necessary, they do not match well with scalable communication architectures where the arbiter is distributed.

Once a particular CPU on the system has exclusive access to the FPU, in *(ii)*, the FPU is configured from that CPU by doing three write transactions, involving the operands A and B, and the opcode of the floating point instruction (see Steps 1-3 on Figure 5.9 on the preceding page).

Finally, from hardware-software communication point of view, in *(iii)*, the notification and its posterior result collection can be performed using different HW-SW methods. The internal architecture (based on the block diagram in Figure 5.2 on page 112) of our Cortex-M1 SP FPU accelerator executes floating point operations at different latencies. The expected latencies for single precision operations are shown in Table 5.1.

Instruction type	**Latency**
Addition	3
Sustraction	3
Multiplication	3
Comparsion	2
Square-root	16
Division	16
Conversions (from/to Float)	3
Mul-Add	6
Mul-Sub	6

Table 5.1: Cortex-M1 SP FPU accelerator core latencies (from [194])

Since our FPU is completely standalone, as we shown in Section 5.5 on the next page, a hardware-dependent software communication library have been implemented to generate a slave-side FPU-CPU notification. This notification will indicate exactly the moment when the FPU has finished to computed the floating point operation, and therefore, the result is ready to be read. In this work, we explored the following hardware-software notification schemes in our Cortex-M1 SP FPU accelerator.

- *No Operation (NOP)* – This mechanism includes NOPs operations on the software library to synchronize wasting CPU cycles exactly the latency required by the FPU core to compute the floating point instruction.

- *Polling* – This mechanism is done from the CPU by polling (performing

read transactions) permanently the *DATA_READY* bit on FPEX register.

- *Interruptions (IRQ)* – A very common slave-side method used when a slave wants to notify the master an issue, in our case, when the result is ready.

The number of transactions of the protocol (see Figure 5.9 on page 117) to perform a floating point calculation can oscillate depending on the availability of the FPU, the HW-SW notification method, and the operation itself. However, when the FPU is in idle condition, the HW-SW protocol takes normally 6 data word transfers.

- 1 read of FPEX register.
- 3 writes to FP Operand A, FP Operand B and FP Opcode registers.
- 1 read of FPSCR register (exceptions and flags).
- 1 read of FP Result A or FP Result B register depending on the floating point instruction.

5.5 FPU Hardware-Dependent Software Library Extensions

Since we have to communicate the Cortex-M1 processor with the FPU, a full hardware-dependent software library have been implemented according the CPU-FPU protocol for each floating point instruction. However, we implement a subset of the most important single precision functions[2]. The implemented functions in our Cortex-M1 FPU software library are *FADDS*, *FSUBS*, *FMULS*, *FDIVS*, *FSQRTS*.

In Listing 5.2 on the next page, we show *FSUB* floating point instruction using our SP FPU accelerator. This function make use of the alternatives CPU-FPU notification protocols described in Section 5.4 on page 116 using pre-compiler directives. FSUB accesses the memory-mapped FPU accerator

[2]Our SP FPU accelerator implements a full-featured single precision floating point functionality, but only a subset of these features have been developed using the proposed hardware-software approach, since the main goal is to explore the performance of the synchronization and the HW-SW protocols under different interconnection fabrics and architectures, rather than test all FP instructions exhaustively.

using a pointer to the memory address where it is placed.

```
unsigned int * AMBA_FPU = (unsigned int *) (FPU_ADDRESS);
```

As shown in Listing 5.2, relative offsets to AMBA_FPU address pointer are used to access FPU registers in the moment to execute floating point instruction.

FSUB and all the other implemented floating point instructions included in our FPU API software library implemented to work with the Cortex-M1 SP FPU accelerator, only generate few assembly instructions, so they are very efficient and they can be executed in few clock cycles.

Listing 5.2: *FSUB* implementation using a HW FPU accelerator

```
unsigned int amba_fpu_fsub(unsigned int a, unsigned int b)
{
  // Getting access to SP FPU accelerator (Step 0)
  volatile unsigned int busy = 1;
  do {
    busy = (AMBA_FPU[FPEX_OFFSET_REG] & FP_EXC_FPU_BUSY_MASK) >> 1;
  } while (busy);

  // Operands and opcode transfers (Step 1-3)
  AMBA_FPU[FP_OPA_OFFSET_REG] = a;
  AMBA_FPU[FP_OPB_OFFSET_REG] = b;
  AMBA_FPU[FP_OPCODE_OFFSET_REG] = FSUBS_OPCODE;

  // CPU waits for FPU result (Step 4)
  #ifdef NOPS_PROTOCOL
    __nop();
    __nop();
    __nop();
  #endif

  #ifdef POLLING_PROTOCOL
    while (!(AMBA_FPU[FPEX_OFFSET_REG] & FP_EXC_DATA_READY_MASK)){}
  #endif

  #ifdef IRQS_PROTOCOL
    while (irqHandler == 0){}
      irqHandler = 0;
  #endif

  // Return the result (Step 5-6)
  return AMBA_FPU[FP_RESULT_PIPEA_OFFSET_REG];
}
```

Using a NoC backbone, the library will use the QoS features to lock/unlock the channel towards the SP FPU. Thus, rather than poll the semaphore on FPU_BUSY bit on FPEX[0] corresponding to Step 0 of the CPU-FPU protocol, the library will substitute it for a call to ni_open_channel(FPU_ADDRESS, TRUE) to open and block the channel

enabling bidirectional traffic towards our SP FPU. This mechanism will add some latency, but will no add any extra traffic on the NoC backbone.

At the end of the function, after reading the result/flags from the FPU (Step 5-6), the function must close the channel, and therefore, it is required a call to `ni_close_channel(FPU_ADDRESS, TRUE)` before the return statement.

The presented software library substitutes the existing `__aeabi_xxxx()` runtime functions which emulates the floating point instructions purely using software instructions. In Listing 5.3, we show the prototypes of the implemented floating point instructions. In case the system includes our SP FPU block, and by using a pre-compiler directive, i.e. `-DCM1_AMBA_FPU`, the ARM compiler will define the `$Sub$$__aeabi_xxxx()`. These functions call directly our equivalent `amba_fpu_xxxx()` hardware FP implementation using our software library.

Thus, at compile time, the compiler will include by attaching the code of our `amba_fpu_xxxx()` macros as a particular implementation of the corresponding floating point instruction. This will enable the hardware assisted execution of the floating point instructions using our proposed HW-SW protocol between the Cortex-M1 and the memory-mapped SP FPU hardware accelerator.

Listing 5.3: HW FPU accelerator software routines

```
// Functions that substitutes the single precision SW emulation routines
// FADD, FSUB, FMUL, FDIV, FSQRT
float $Sub$$__aeabi_fadd(float a, float b);
float $Sub$$__aeabi_fsub(float a, float b);
float $Sub$$__aeabi_fmul(float a, float b);
float $Sub$$__aeabi_fdiv(float a, float b);
float $Sub$$_fsqrt(float a);
```

Despite this fact, the original software library that only executes floating point instructions purely in software can be also called using `%Super%%__aeabi_xxxx()` corresponding to each FP operation (see Listing 5.4 to see the prototypes).

Listing 5.4: Floating point software emulated routines

```
// Functions that call directly the single precision
// software emulated routines in case you needed
float $Super$$__aeabi_fadd(float a, float b);
float $Super$$__aeabi_fsub(float a, float b);
float $Super$$__aeabi_fmul(float a, float b);
float $Super$$__aeabi_fdiv(float a, float b);
float $Super$$_fsqrt(float a);
```

5.6 Experimental results

In this section, we report the results obtained to implement our SP FPU in a real prototyping platform (Xilinx and Altera FPGA devices). Later, we integrate the SP FPU on a complete Cortex-M1 MPSoC cluster architecture presented in Figure 5.10 on page 124, which can be parameterized using two interconnection fabrics, and the different alternatives CPU-FPU protocol to communicate the operation result.

Similarly, we include on the software stack of this Cortex-M1 MPSoC platform, the floating point API software library to enable hardware-assisted execution of floating point instructions.

The proposed HW-SW framework, let us *(i)* to assess and explore the validity of our HW-SW components; *(ii)* to compare our standalone memory-mapped SP FPU against the tight-coupled ones (like the one provided by Xilinx); and *(iii)* the exploration of the ability to share our SP FPU in terms of execution time according to different synthetic floating point workloads.

The exploration also reports speedups between our hardware-assisted FP execution versus a pure software emulation of floating point instructions. Finally, we compute an significant metric, such as the MFLOPs, which is very important to ensure high-performance computing on embedded systems.

5.6.1 Cortex-M1 SP FPU Hardware Synthesis

In this section we report, the synthesis results, in terms of LUTs, f_{max}, number of embedded memory and DSP blocks used for our Cortex-M1 SP FPU accelerator. We synthesized the proposed SP FPU accelerator in multiple FPGA devices to explore the trend by evaluating the performance and area costs. Altera and Xilinx are currently the main FPGA vendors, and since ARM Cortex-M1 and our SP FPU accelerator have been targeted to be used in FPGAs, we are going to compare our memory-mapped SP FPU accelerator with the equivalent tight-coupled one used by Xilinx PowerPCs.

In Table 5.2 on the next page, we show the area costs and the f_{max} of Xilinx FPU on Virtex-5 FXT FPGA used in PowerPC processors (see [190] for further information). In Table 5.3 on the facing page, the results of our Cortex-M1 SP FPU accelerator[3] are reported using the same terms in multiple Xilinx and Altera FPGAs.

[3]Even if our SP FPU accelerator has been considered to be used with Cortex-M1 soft-core processor, thanks to its AHB compatible interface, it can be used for any other processors or systems that follows AMBA 2.0 AHB standard.

	Part/Device	Speed (MHz)		LUTs	rams	DSP Blocks
		C_LATENCY=0	C_LATENCY=1			
Double Precision	Virtex-5	200-225*	140-160*	4950	0	13
Single precision	Virtex-5	200-225*	140-160*	2520	0	3

* Depending on Virtex-5 FXT speed grade (-1 or -2), f_{max} range goes from 200-225 MHz and 140-160 MHz according to the FPU configuration (C_LATENCY=0 for high-speed and C_LATENCY=1 for low latency execution of floating point instructions.

Table 5.2: Synthesis of single and double precision PowerPC FPU

Focusing on the single precision implementation, we can affirm that our Cortex-M1 SP FPU accelerator (see Table 5.3) is very competitive in contrast with the provided by Xilinx (see Table 5.2). Thus, in our implementation we use only 270 LUTs more (2520 for Xilinx FPU vs. 2790 of our Cortex-M1 SP FPU accelerator). This overhead probably may be caused by the integration of additional blocks in our standalone memory-mapped FPU, such as, the register bank, the semaphore, the AMBA 2.0 AHB interface and its decoder unit, and the external multipliers, etc, which presumably are not counted in the Xilinx implementation. In terms of f_{max}, our SP FPU is only 5% slower compared to the Xilinx FPU (134.2 MHz vs. 140 MHz) on a Virtex-5 FPGA, since our SP FPU is optimized to be low-latency (C_LATENCY=1). Obviously, depending on the FPGA technology and speed grade the circuit performance oscillates. For instance, on lasts Altera Stratix III devices the FPU can reach 176.5 MHz, whereas when the synthesis is done on an Stratix I FPGA, the frequency drops until 76.9 MHz.

	Part/Device	Speed (MHz)	LUTs	rams	DSP blocks
Xilinx	xc2v1000bg575-6	84.0	3592	1 M512	4
	xc4vfx140ff1517-11	101.6	3655	1 M512	4
	xc5vlx85ff1153-3	134.2	2890	1 M512	4
Altera	EP1S80B956C6	76.9	3937	0	1
	EP2S180F1020C4	132.0	2697	0	1
	EP3SE110F1152C2	176.5	2768	0	1

Table 5.3: Cortex-M1 SP FPU accelerator synthesis on Xilinx and Altera FPGA devices

Finally, our SP FPU design on Xilinx FPGAs use 4 blocks DSP48 against the 3 used by PowerPC FPU, and it requires 1 M512 block of embedded RAM to map the FPU register bank. Surprisingly, on Altera FPGAs, our FPU does not require embedded FPGA memory (the register bank is implemented using LUTs and registers) and it only needs one DSP block.

As shown in the above tables, the area of the FPU is quite big (around 2x the area of Nios II [20] or MicroBlaze [18] soft-core processors, and 4x bigger when double precision support is included). Since our FPU design is completely standalone and memory-mapped as an accelerator on the system, rather than tightly-coupled with the CPU, we can think to share an arbitrary number of FPUs among different CPUs. To explore this possibility, and to validate the presented HW-SW infrastructure, a complete framework have been developed, integrating the software API library to perform hardware-assisted floating point operations in conjunction with our SP AMBA 2.0 AHB compliant FPU. For comparison, as shown in Figure 5.10, we have integrated our SP FPU on an AMBA 2.0 AHB template Cortex-M1 MPSoC architecture, which can be designed using two different communication backbones[4].

- An AMBA 2.0 AHB Multi-Layer [27].
- A NoC backbone generated by ×pipes [52, 53] and using the AMBA 2.0 AHB NIs presented in Chapter 3 on page 57.

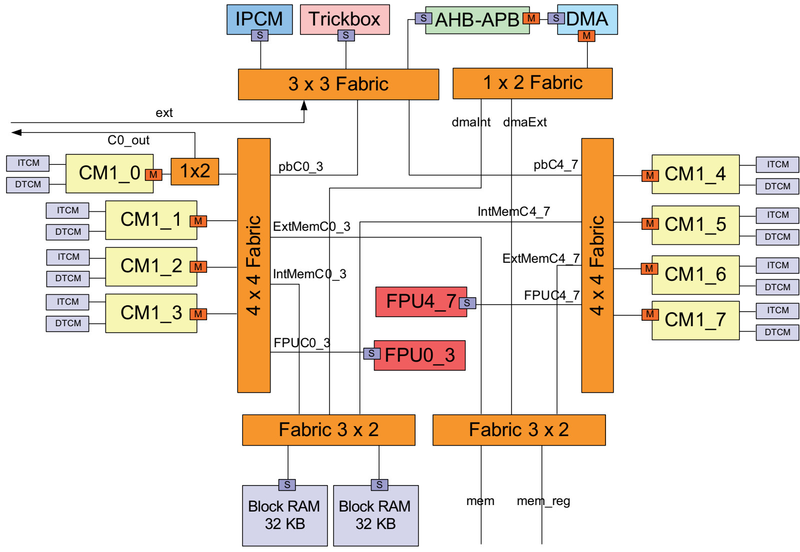


Figure 5.10: Cortex-M1 MPSoC cluster architecture

This template architecture presented in Figure 5.10 consists on two symmetric 4-core Cortex-M1 banks interconnected by using one of the previous

[4]Both communication fabrics have been configured to operate at a single clock frequency.

interconnection fabrics. In each bank, we integrate our SP FPU, resulting that each bank shares a single FPU among the four Cortex-M1 processors. Each Cortex-M1 consist of:

- ARM-v6M architecture compatible core.
- 3-stage pipeline core.
- L1 memory system providing tightly-coupled instruction and Data memories (DTCM/ITCM).
- Interrupt controller supporting 32 interrupts levels.
- 32-bit Timer.

The MPSoC architecture also contains a L2 shared 64-Kbyte AHB scratch pad AHB memory to provide efficient data and program buffering. Finally, the external memory is L3 memory.

An eight channel *DMA* controller to support transfers between Cortex-M1 TCM and the AHB scratch pad or the external memory, or even between TCMs of the cores in the same cluster is also included. The system also supports message passing by using *Inter-Processor Communication Module (IPCM)* modules between cores in the MPSoC. A *Trickbox* is used to control asynchronous inputs signals to the cores, for instance generating reset pulse to keep processors in idle, but also to check the status of the cores (e.g. halted or lockup) is implemented within this block. Finally, a master and slave AHB port is provided to enable data transfers with another external systems and subsystems.

In Table 5.4 and 5.5 on the following page, we show synthesis results of the Cortex-M1 MPSoC using an AHB multi-layer as a communication backbone, with and without floating point support. This architecture has been configured to use 8 KB for DTMC/ITCM, and 32 KB for scracthpads AHB memories.

The impact in terms of circuit performance and area resources to integrate 2 FPUs to be shared for each bank of 4 Cortex-M1 processors is not negligible but it can be assumed. The f_{max} is not strongly affected with the inclusion of the FPU, and surprisingly in some cases the resulting f_{max} is even better when the system includes floating point support in hardware. This is because the critical path of the system is in the memory controller rather within the SP FPU block. In terms of area, the overhead to include the proposed shared SP FPU among each bank of 4 Cortex-M1s represents about 12-14% of the whole system depending on the FPGA technology.

Part/Device	Speed (MHz)	LUTs	rams	DSP blocks
xc4vfx140ff1517	70.0	50315	RAM16: 512 Block RAMs: 98	24 DSP48
xc5vlx85ff1153	117.6	38294	RAM128X1D: 512 Block RAMs: 98	24 DSP48
EP2S180F1020C4	90.9	34360	M4Ks: 400 M512s: 16 ESB: 1613824 bits	8 (78 nine-bit DSP)
EP3SE110F1152C2	122.3	34696	M9Ks: 226 ESB: 1613824 bits	4 (32 nine-bit DSP)

Table 5.4: Synthesis of Cortex-M1 MPSoC cluster without hardware floating point support on Xilinx and Altera FPGAs

Part/Device	Speed (MHz)	LUTs	rams	DSP blocks
xc4vfx140ff1517	74.4	57346	RAM16: 512 Block RAMs: 98	32 DSP48
xc5vlx85ff1153	123.9	43790	RAM128X1D: 512 Block RAMs: 98	32 DSP48
EP2S180F1020C4	87.8	39979	M4Ks: 400 M512s: 18 ESB: 1614592 bits	10 (78 nine-bit DSP)
EP3SE110F1152C2	124.1	39936	M9Ks: 226 ESB: 1614592 bits	6 (46 nine-bit DSP)

Table 5.5: Synthesis of Cortex-M1 MPSoC cluster with hardware floating point support on Xilinx and Altera FPGAs

On the same MPSoC architecture (see Figure 5.10 on page 124), where each processor has its own tight-coupled FPU, the overhead will be considerable, and only the dedicated area resources on the system to support floating point in hardware will increase up to 35-39%.

5.6.2 Performance Studies – HW-FPU accelerator vs. SW Emulation Floating Point Execution

In order to validate the designed HW and SW components, we run some synthetic FP benchmarks or synthetic μkernels in multiple scenarios varying the interconnection fabric, and the CPU-FPU synchronization and notification protocols.

First, to test the performance, a single mono-processor architecture has been used using with only one Cortex-M1 processor connected to our SP

FPU accelerator through an AHB ML. Figure 5.11 reports performance in our Cortex-M1 SP FPU accelerator, measured in clock cycles, for our different CPU-FPU notification protocols[5].

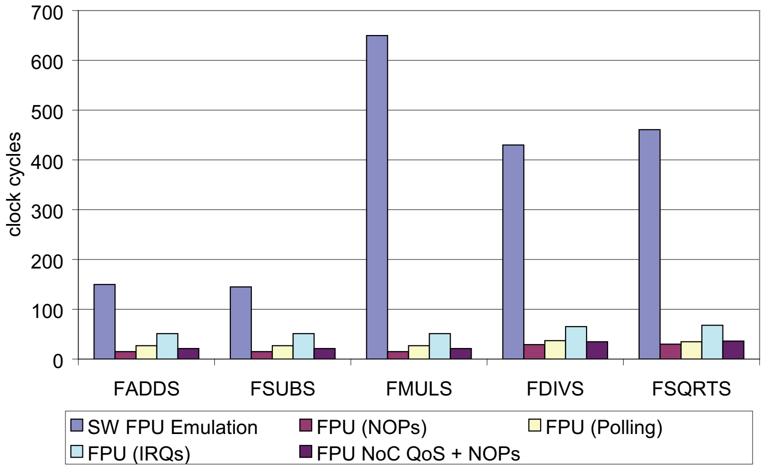


Figure 5.11: Cortex-M1 SP FPU accelerator execution times of various floating point operations using different CPU-FPU notification methods vs. SW emulation

When NOPs are used, we achieve the maximum performance, i.e. between 15-30 clock cycles to compute FP operations, since the Cortex-M1 and the FPU are synchronized ideally. This is because we include exactly the same number of NOPs according to the latency of the single precision core (see Table 5.1 on page 118).

Similarly, when a polling CPU-FPU notification protocol is used the performance is a little bit worse compared with NOPs. This is because polling often penalizes because of its high bus usage, and the de-synchronization between the time when the result is ready on the FPU, and the one when the CPU realizes of it. However, this notification protocol has similar performance to NOPs, and the latency of the hardware core can be changed independently from the software library.

Finally, if interruptions are used, even if Cortex-M1 is optimized to perform them, the notification protocol spend almost the double of clock cycles than NOPs or polling. We explain this effect by observing the context switching and the IRQ handler routines, which takes around 30 cycles.

[5]The benchmark, as well as the implemented hardware-depedent software library have been executed using the tight-coupled memories, which load one instruction every cycle.

However, in contrast to polling notification, no traffic is added on the communication backbone, resulting on a reduce bus utilization.

We extended our exploration on an equivalent NoC-based MPSoC architecture using an equivalent NoC backbone as the AHB multi-layer presented in Figure 5.10 on page 124. In these experiments, we fix the CPU-FPU notification protocol using NOPs. The proposed scheme using NoC transactions and QoS features case exhibits a performance level close to the one using in the AHB ML. Basically, it takes few more cycles than NOPs due to the overhead to open/close the channel to get exclusive access to our Cortex-M1 SP FPU block.

To validate the effectiveness of our approach, we compare our hardware-assisted framework to compute FP operations using our SP FPU accelerator against a purely software emulation. In Figure 5.11 on the preceding page, it is very clear that while the FPU accelerator takes tens of cycles, the software emulation consumes hundreds cycles. Additionally, in Figure 5.12, we show exactly the speedup of each implemented FP operation. Depending on the CPU-FPU and the FP operation, the speedups range from $\sim$3x to $\sim$45x compared to a purely software emulation.

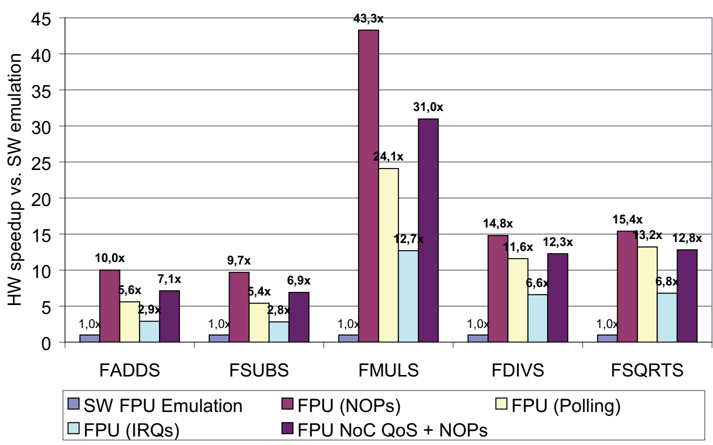


Figure 5.12: Cortex-M1 SP FPU accelerator speedup vs. SW emulation FP library of different floating point operations

Second, to test the shareability and the resulting performance to execute FP operations, we try to vary the number of processors which access the FPU on the Cortex-M1 MPSoC cluster architecture. Scalability results are shown in Figure 5.13 on the facing page, in terms of execution time of a running a synthetic stress test by doing one of the 5 implemented FP operations.

Similarly, in Figure 5.14, we report results in terms of speedup, variating the number of Cortex-M1 CPUs from 1 to 4 (using only half of Cortex-M1 MPSoC cluster) against its execution using software emulation. The plot shows that as the traffic on the interconnection fabric increase, resulting on access congestion (by doing semaphore synchronization or by blocking channels when NoC backbone) completion latencies, and as a consequence, the resulting speedups decrease smoothly according each CPU-FPU protocol.

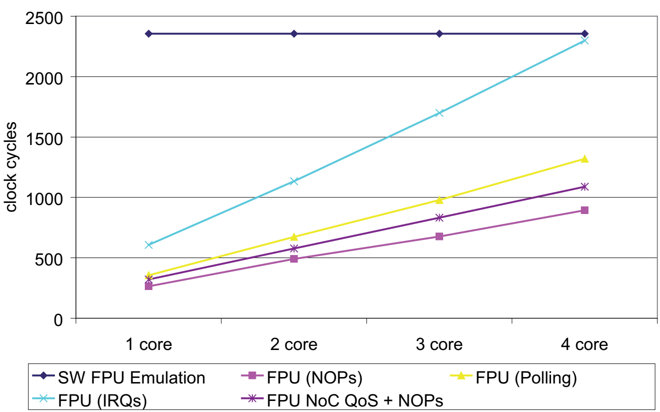


Figure 5.13: Cortex-M1 SP FPU accelerator execution times using different CPU-FPU notification protocols

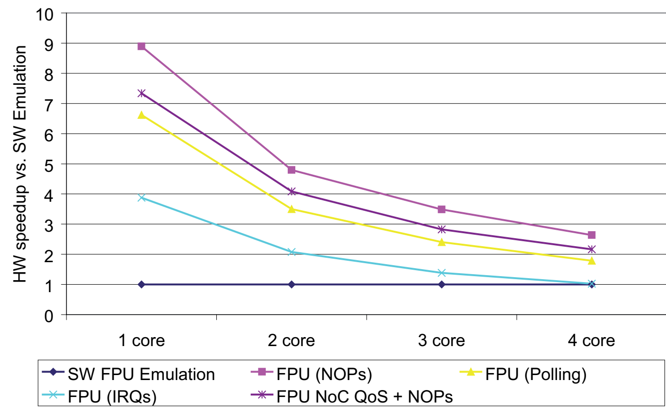


Figure 5.14: HW FPU accelerator speedup vs. SW emulation on different Cortex-M1 MPSoC

Experimental results show that, in presence of concurrent tasks running floating point instructions on multiple processors, the shareability of the SP

FPU has a bound. As shown in Figure 5.13 and Figure 5.14 on the preceding page, under our FP workloads stress tests, the sharing ratio is 1:4, so each bank of 4 processors uses 1 FPU accelerator, otherwise, the use of software emulation performs better, and it does not make sense to integrate more FPUs on the MPSoC architecture.

As a consequence, the proposed Cortex-M1 MPSoC cluster architecture delivers between 1.1-3.5 MFLOPs at only 50 MHz, depending on the CPU-FPU notification protocol used.

Of course, all results shown in previous plots are relatively slow when we compare our SP FPU accelerator against a tight-coupled FPU. This is mainly based on the fact, that our design does not support pipelined FP instructions, since the FPU accelerator is memory-mapped on the bus, and we use a simple register bank. As a consequence, early estimations, show that our SP FPU accelerator is ∼5x slower than a tight-coupled one.

5.7 Conclusion and Future Work

This chapter describes the complete design of an AMBA 2.0 AHB standalone memory-mapped SP FPU, and the hardware-dependent software library used to accelerate single precision floating point functions. The library can be used to speedup floating point computations. The HW-SW framework will lead to achieve high-performance computing on an embedded mono-processor or multi-core ARM Cortex-M1 cluster architectures on FPGAs, or any other system and processors which use AMBA 2.0 AHB standard.

The presented approach can be used in many widely digital multimedia and signal processing algorithms and kernels to take advantage to execute FP instructions in hardware whenever is required. Our SP FPU accelerator will lead to speedup FP operation to achieve high-performance embedded computing on mono-processor or multi-core ARM Cortex-M1 cluster architectures, using a moderate amount of hardware resources (2890 LUTs and few DSP blocks on FPGA devices).

In addition, the methodology for modifying software applications have been done without any change on the ARM compiler and, by extension to the ARM-v6M architecture, so that it is completely transparent to the software designer.

On the other side, we also explore different HW-SW interfaces on the execution platform (see the abstract view presented in Figure 5.15 on the next page) in order to get an efficient execution of hardware-assisted floating point instructions, using our hardware-dependent software routines. Thus, as shown in the experimental results, depending on the system require-

ments, different CPU-FPU notification protocols can be used in our FPU accelerator. The outcome of our exploration, shows that NOPs and polling outperforms the IRQ CPU-FPU notification protocol because of the overhead added by the switching context, and the IRQ handler. However, on heavily traffic conditions, when other peripherals required to use the communication backbone, IRQs can perform better than polling, since it does not inject any traffic on the intercommunication architecture.

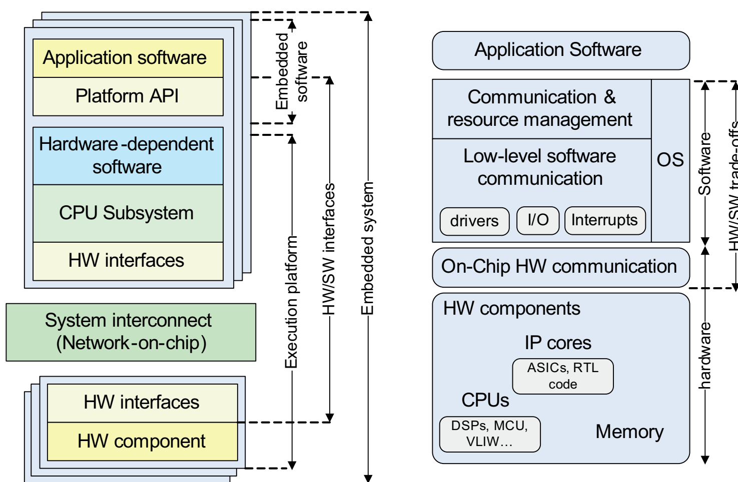


Figure 5.15: HW-SW interface co-design on embedded SoC architectures (from [43])

The exploration to share one FPU among few processors, has been demonstrated as a feasible alternative when a low-cost constraints are imposed, and of course, when the application does not access very frequently and simultaneously to the FPU accelerator. In our experiments, we performed several stressed FPU tests in our Cortex-M1 cluster on chip, and we found that the bound to achieve better execution times that the equivalent software library is about 1:4, i.e. 1 FPU accelerator every 4 ARM Cortex-M1 processors.

As future work, the first engineering task is to extend the software library to use the remaining single precision FP instructions, which are already implemented on the designed AMBA AHB single precision FPU accelerator. The second one, since the designed FPU is targeted to be the FPU of the Cortex-M1 on FPGAs, it would be convenient to tweak the RTL code to map properly the design on the FPGA architecture to improve area cost, and the circuit performance (i.e. f_{max}) of the SP core using properly embedded shifters, DSP blocks, etc. Finally, using the presented HW-SW design approach, it should be useful to design an equivalent double precision FPU

accelerator to speedup operations on C/C++ doubles.

In terms of research, some extension and improvements on the register bank can be done, including an accumulator register bypassed from the result of the last operation to perform single and double compound FP operations, such as fused multiply-add/sub (FMAC, FNMAC, ...).

Additionally, to enable FP instruction pipelining (as it is done when the FPU is tight-coupled with the processor) to improve a bit more the performance of our standalone memory-mapped FPU, an extension of the register bank adding `NextOperand A`, and `NextOperand B` registers. Another potential improvement will be the possibility to decouple the pipeline architecture of the FPU in order to execute simultaneously adding/subtraction with multiplications/divisions/conversions/square, since both pipelines are independent. This will imply deep changes on the front-end of the FPU, duplicating the register bank for each pipeline, and re-designing the policy to block/unblock the access, specially when more than one processor pretend to execute floating point operations on it.

CHAPTER 6

Parallel Programming Models on NoC-based MPSoCs

This chapter[1] introduces parallel programming models focusing on MP-SoCs architectures. Next, we present our message passing software stack and API library (i.e. ocMPI/j2eMPI), and its efficient interaction with Network Interfaces (NIs) to improve inter-process communication through message passing. This chapter also contributes on the mapping of QoS features presented in Chapter 4 on page 85, on top of a custom OpenMP runtime library targeted to NoC-based MPSoCs, or event on the parallel programming model at application level.

6.1 A General Overview of Parallelism

There is an on-going requirement to solve bigger and more realistic problems, and this requires improved computer productivity. This increase can arise through faster processing power, specialized computers (where some extra capability is included in the computer hardware), or by writing parallel code. In parallelism exists two levels of abstraction. *Fine-grain parallelism* refers to parallelism at the operand or data level (e.g. at pixel level, matrix rows and columns, etc), where parts of non-dependent data can be processed concurrently, or high-level parallelism, whereas *coarse-grain parallelism* refers to entire routines being executed concurrently (e.g. frame, DCTs, FFTs, etc).

An important feature of a parallelism is the algorithm scalability; this

[1] The author will like to acknowledge contributions by Andrea Marongiu, and Prof. Luca Benini for its contribution on the OpenMP framework for embedded systems.

means that the level of performance is maintained as the number of processors and the problem size are increased. Another challenge is how the application is broken down to individual parts or tasks, and how these tasks will be mapped onto processors, and which cost is inherent in their communication. In addition, in most cases it should be mandatory to use load-balancing techniques, and study the locality between each task as well on data (i.e. maximizing the reuse of data in nearby locations, by minimizing the number of accesses to data in far locations) and their relationship in order to obtain better performance by minimizing the communication and overlapping computation and communication. Thus, using these considerations ensure that all the processor in the resultant parallel system will work concurrently rather than some processors being idle, and it lets that closer tasks collaborate without large communicate overhead.

Once we introduced the basics of parallelism, next are detailed the existing ways of parallelism together with several classifications that have been proposed in order to structure it in logical categories, for example, according to the architecture memory model, processor architecture or other recently proposed levels of parallelism.

The first traditional classification is Flynn's taxonomy [195]. It categorizes the various architectures of parallel computers, although this has now been extended to distinguish between shared memory and distributed memory systems:

- Single Instruction, Single Data (SISD): consists of a single CPU executing an instruction stream serially.
- Single Instruction, Multiple Data (SIMD): these machines often have many processors that execute the same instruction all the time but on different data; they operate in lock-step mode. A subclass of SIMD machines are vector processors, which operate simultaneously on an array of data. This is fine-grained parallelism, with little inter-processor communication required (e.g. VLIW, DSP).
- Multiple Instructions, Single Data (MISD): as yet, no machine with this classification has been developed.
- Multiple Instructions, Multiple Data (MIMD): these machines operate in parallel on different data, using different instructions. There is a wide class of classification, and now this is often broken down into either shared-memory or distributed-memory systems.

Nowadays, *Single Program Multiple Data (SPMD)* and *Multiple Program Multiple Data (MPMD)* are widespread, but whereas MPMD is becom-

ing popular, SPMD is the most used model. In this model, multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that SIMD imposes on different data. With SPMD, tasks can be executed on general purpose CPUs; SIMD requires vector processors to manipulate data streams. SPMD usually refers to message passing programming on distributed memory computer architectures. The first SPMD standard was PVM [196], but the current de-facto standard is MPI [142].

SPMD on a shared memory machine (a computer with several CPUs that access the same memory space), messages can be sent by depositing their contents in a shared memory area. This is often the most efficient way to program shared memory computers with large number of processors, especially on NUMA machines, where memory is local to processors and accessing memory of another processor takes longer.

According to previously presented memory models, the parallelism can be classified in two categories (see Table 6.1 on the next page):

- UMA (Uniform Memory Access): in this architecture, the memory access time is independent from which processor makes the request or which memory contains the requested data. Usually, this architecture is used on SMPs, which normally includes a global shared memory for all processors.
- NUMA (Non-Uniform Memory Access): this architecture is used in multiprocessors where the memory access time depends on the relative memory location to a specific processor. An example of a shared-memory NUMA is cache coherence NUMA architecture. Under this architecture, any processor can access its own local memory faster than non-local memory, in most cases shared by all processors. In shared-memory systems, we must synchronize the access to shared data, for example using semaphores. Opposite to NUMA with shared memory, are the distributed memory systems. These systems are also called massive parallel processing (MPP) systems because many processors are used to work together in a distributed way. In these systems each processor has its own local memory, and all processors are connected through a network (which needs to be well-designed to minimise communication costs). Therefore, these systems need to use message-passing techniques to share and communicate data among them.

The main disadvantage of UMA architectures against NUMA is the scalability. In other words, under UMA architectures when number of proces-

Memory model	Parallel Architecture	
	UMA	NUMA
Shared memory	SMP	ccNUMA (cache coherence NUMA)
Distributed memory	Not used	MPP (Massive Parallel Processing

Table 6.1: Parallel architectures according to memory models

sors increases over a threshold the system does not scale well. Another important aspect is that both SIMD and MIMD can use either shared or distributed memory models.

A part from this traditional classification, more types of parallelism emerge nowadays in research. One is *Instruction Level Parallelism (ILP)* that is a measure of how many operations in a computer program can be performed simultaneously. Techniques that exploit this type of parallelism are related to the processor micro-architecture. The compiler and the processor are basic pieces to obtain as much parallelism as possible. The most important techniques are:

- Instruction pipelining: the execution of multiple instructions can be partially overlapped

- Superscalar execution: multiple execution units are used to execute multiple instructions in parallel. In superscalar processors, the instructions that are executed simultaneously are adjacent in the original program order.

- Out-of-order execution: instructions can be executed in any order that does not violate data dependencies. Out-of-order execution is orthogonal to pipelining and superscalar since it can be used with or without these other two techniques.

- Register renaming: technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers. This technique is usually used with out-of-order execution.

- Speculative execution: allows the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is on flow control instructions (e.g. a branch). A pre-fetch is executed before the target of the control flow instruction to determine which branch will follow the execution. Several other forms of speculative execution have been proposed and are in use, including those

driven by value prediction, memory dependence prediction and cache latency prediction.

- Branch prediction: used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.

Other important type of parallelism is *Thread Level Parallelism (TLP)*. This parallelism is important due to the fact that multithreaded processors are emerging in the design of generic multiprocessors, and in specific MPSoC. An example is the CMT Niagara processor [197–200]. TLP is based on parallelism inherent to the application that runs multiple threads at once. Thus, while one thread is delayed, for example, waiting a memory access, other threads can do useful work. The exploitation of coarse-grain TLP, inherent to many multimedia algorithms, has not been investigated in depth up to now because uniprocessor model dominated the research community.

6.2 **Traditional Parallel Programming Models**

The presented traditional parallel architectures explained in previous section (shared memory or distributed memory systems) imply the use of different parallel programming models.

- Message Passing: a message passing scheme is required to perform data communication, synchronization among all processors through the communication infrastructure, which should be very efficient to ensure low-latency inter-processor communication, synchronization, communication or task partitioning.

- Shared memory: In shared memory systems is necessary to ensure the coherence, synchronization, and memory ordering. The cache coherence model is applied, i.e. whenever one cache is updated with information that may be used by other processors, then all the changes must be reflected to the other processors in order to get coherent data. Sometimes this model can become a performance bottleneck of performance because of the overhead to exchange the information related to the memory coherence (e.g. snooping, write-through, write-back).

Therefore, several software runtime environments and standard software APIs have been proposed according to these well-known programming models. The most significant are *Parallel Virtual Machine (PVM)* [196, 201] and *Message Passing Interface (MPI)* [142, 202–204] for message passing

on distributed-memory systems, and *OpenMP* [205] for shared-memory systems.

The target is to provide software developers libraries to express concurrency and parallelism on many different ways in a great variety of parallel platforms to perform an efficient inter-process communication between all processes, cores and threads.

Both parallel programming models and API-libraries abstract the hardware and software components and let the designer to execute concurrent or parallel applications in a multiprocessors system, and at least they impose two general ways to express parallelism:

(i) Implicit parallelism: a compiler or interpreter exploit the parallelism inherent to the computations expressed by some of the language's constructs. It is very easy to program but sometimes sub-optimal results can be achieved because the programmer does not have the complete control over the parallelism, and significant hardware support (i.e. cache coherence) must to be added to achieve consistency on the parallelism.

(ii) Explicit parallelism: the programmer has the control over the parallel execution, so a high-skilled parallel programmer takes advantage of it to produce very efficient parallel code. However, it is difficult for unskilled programmers because many special-purpose directives or functions, mostly for synchronization, communication or task partitioning, must be called by the programmer. This problem can be mitigated using optimizing compilers or communication assistants.

Often, OpenMP is associated to *implicit parallelism* due to the fact that the inclusion of `#pragma` compiler directives let programmers to express parallelism using a single-threaded view. On the other side MPI/PVM is related to *explicit parallelism* since software developers must explicitly call the communication and synchronization routines to express parallelism.

Recently, a novel parallel programming model based on database transaction processing have been proposed in the form of *Transactional Memory (TM)* [206–209]. Transactional memory is aimed as a mechanism for implementing language-level concurrency control features to speed up sequential code transparently from the user point view by exploiting the thread-based programming [210], and the inherent parallelism of multithreaded architectures on top of cache coherence model. It simplifies parallel programming by guaranteeing that transactions appear to execute serially (i.e. using a consistency model to ensure commit order) and atomically (i.e. all or nothing,

a transaction never appear to be interleaved with another transaction) or isolated (i.e. write are not visible until transaction commits). Thus, a transaction either commits by making all changes visible to the other processes, or abort, by discarding all the changes doing a roll-back process.

This parallel programming model is nowadays under research. Nevertheless, the feasibility of this approach is unclear due to the large traffic overhead generated at HW/SW level (i.e. snooping and roll-back process using validate, commit or abort transactions) in order to serialize multiple threads of the different cores.

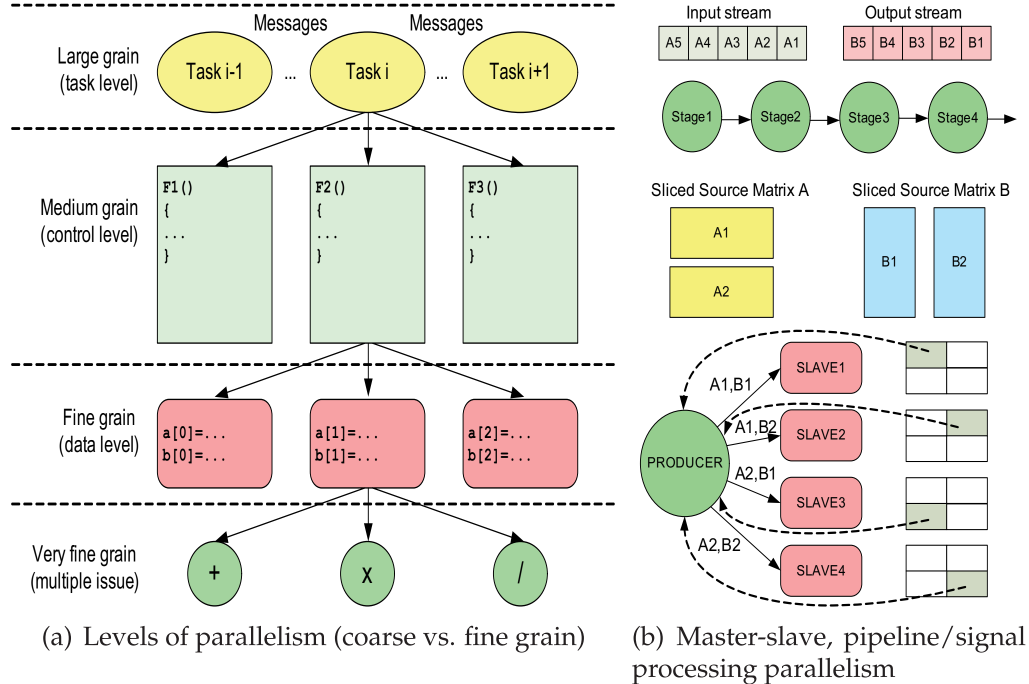
6.3 Motivation and Key Challenges

Moore's law is still alive but the technology clock frequency scaling has been replaced by scaling the number of cores per chip as is reported in ITRS [1]. In addition, performance has also slowed along with the power, in all systems, traditional multiprocessors or MPSoCs. Thus, nowadays each core operates at 1/3 to 1/10th efficiency of largest chip, but you can pack hundred very simple cores onto a chip and they consume less power. Thus, in today's MPSoCs, in conjunction with an extremely low-cost interconnection infrastructure previously discussed in this dissertation, parallel programming models are needed to fully exploit hardware capabilities and to achieve hundred GOPS/W energy efficiency.

The mapping of this abstract parallel programming models onto tightly power-constrained hardware architectures imposes several overheads which might seriously compromise performance and energy efficiency. Thus, an important key challenge in MPSoC research is to provide well-know custom parallel programming models and libraries to boost applications by exploiting all hardware features present in the platform, but taking into account the inherent memory constraints present on NoC-based MPSoC architectures. Using parallel programming models hide hardware complexity from the programmer point of view, facilitating the development, reusability and portability of parallel applications.

The goal to execute parallel applications is to reduce the wall clock time (speed-up) saving time on the resolution of large problems exploiting the HW features present in the parallel platform. A key challenge is how the problem is mapped on hardware and broken into concurrent streams of instructions on the multiprocessor platform. Therefore, the software APIs or the compiler directives associated to the parallel programming model should offer facilities to split effortlessly data, functions or high-level tasks in many different ways.

In fact, using the parallel programming models presented before, many different levels and mechanism of parallelism can be reached. In Figure 6.1, we show the most common ones.



(a) Levels of parallelism (coarse vs. fine grain)

(b) Master-slave, pipeline/signal processing parallelism

Figure 6.1: Multiple levels and ways of parallelism

Usually, MPI is related to parallelism at large medium grain task level exchanging messages (see Figure 6.1(a)) specially with computation intensive applications with large data sets. However, it can be used to enable medium and often in fine grain parallelism at function and data level, as well as in synchronization-intensive applications with small data sets. On the other hand, OpenMP is very useful to low computation/communication ratio emphasizing shared memory efficiency. Normally is used from task level parallelism until data level parallelism. Finally, very fine grain parallelism can be achieved at operator and instruction level using custom VLIW, DSPs, but it is out of the scope of this dissertation, which is focus on the use of parallel programming models.

Both MPI and OpenMP define multiple ways how a parallel workload is assigned to the parallel system to be performed simultaneously by different processor cores. In Figure 6.1(b), we show master-slave (i.e. broadcast-reduce, scatter-gather, etc) and pipelined or signal processing schemes when a workload is parallelized on different computations sub-

tasks. However, MPI suits perfectly in both schemes, in OpenMP is not straight to perform an efficient pipeline due to memory contention.

The use of parallel programming models must be supported by robust software stacks and efficient HW-SW interfaces to avoid large overheads of inter-process communication and synchronization. Thus, performance is now a software concern, so the compilers directives or API libraries, which abstracts hardware and software components must be designed accurately. In addition, the responsibility and the performance is shifted to programmers, so parallel programming models must expose features to deal with performance.

Finally, at the moment to develop or choose an existing parallel programming model, many considerations must be taken into account:

(i) *Parallel Execution Model:* implicit or explicit communications using synchronization and/or memory consistency methods.

(ii) *Productivity:* is related to the language or APIs used to express parallelism. If a new language is provided, developers must learn it. On the other side, the use of an existing language will not need the reeducation of developers. An hybrid approach is to extend (limited by the constraints of the base language) or provide directives (to achieve higher levels of abstraction) for an existing language to match the platform requirements. In the best case, developers only need to know very few new API constructs.

(iii) *Portability:* is associated to the runtime system and its relation on specific machine features.

(iv) *Performance:* latency management, static and dynamic load balancing are key features to be exposed to the programmer in order to enable efficiency of parallel programs.

(v) *Debug tools:* parallel programming environment should built-in proprietary tools in the runtime library, provide logs to enable tracing and performance evaluation (e.g. Paraver [211], Vampir [212, 213]) or enable interoperability with standard tools (e.g. gdb, grprof) available on the platform because debug parallel programs is tough.

6.4 Related work

Due to the necessity to program homogeneous and heterogeneous MPSoCs to increase performance/power figures, recently a large number of MP-

SoC specific programming models and software APIs have been defined or ported, for stream processing, shared-memory or message-passing parallel programming models.

The Task Transaction Level interface (TTL) proposed in [214] focuses on stream processing applications in which concurrency and communication are explicit. The TTL API defines three abstraction levels: the vector read and vector write functions are typical system level functions in which synchronization and data transfers are combined: the *reAcquireRoom* and *releaseData* functions. In [215], a compiler and a novel high-level language named Stream-It, in order to provide syntax to improve programmer productivity for high-performance streaming applications is reported.

In [216], the Multiflex multi-processor SoC programming environment is presented. It is focus on two programming models: a remote procedure call on distributed system object component (DSOC) message passing model, and a symmetrical multi-processing (SMP) model using shared memory (close to the one provided by POSIX [217], i.e. thread creation, mutexes, condition variables, etc). It is targeted to multimedia and networking applications, with the objective of having good performance even for small granularity tasks. DSOC uses a CORBA like approach, but implements hardware accelerators to optimize performance. In [218], a Remote Method Invocation (RMI) model is explored to communicate distributed heterogeneous NoC-based systems.

The transactional memory model also have been implemented in MPSoCs. In [219] is reported a complete hardware transactional memory solution for an embedded multi-core architecture using MPARM [54], consisting of a cache-coherent ARM-based cluster on a shared-memory system, similar to ARM's MPCore [220], meanwhile in [221] the idea on how to design a lightweight transactional memory system is presented.

The Eclipse NoC architecture is presented in [222] in conjunction with a sophisticated programming model, realized through multithreaded processors, interleaved memory modules, and a high-capacity NoC. Eclipse provides an easy-to-program, exclusive-read, exclusive-write (EREW) parallel random-access machine (PRAM) programming model with many physical threads to take profit of TLP.

Recently, a large body of research efforts have been done to fit shared-memory and message passing parallel programming models upon MPSoCs architectures. Thanks to its portability both, OpenMP and MPI APIs can be adapted effortlessly keeping the interface invariant but being improved by custom hardware features.

In [223] is reported the use of OpenMP targeted to shared-memory MP-

SoCs without using any operating system, or OpenMP extensions, but improving low-level synchronization directives. In [224] OpenMP is ported on top of the Cell Broadband Engine™ processor [25]. This work presents thread and synchronization support, code generation and the memory model on Cell. The compiler is novel and it generates binaries that can be executed across multiple heterogeneous ISAs and multiple memory spaces. In [225] is presented another OpenMP approach targeted for heterogeneous MPSoCs, which includes RISC-like processors and DSPs, using OpenMP extensions. Recently, in [183] the authors also explore extensions and the heterogeneity by optimizing the OpenMP implementation and the runtime support, for non-cache-coherent distributed memory MPSoCs. This work uses extensions to perform a profiled-based efficient data allocation with an explicit management of scratchpads memories.

In addition, together with all these efforts, in [226, 227] explain the need to abstract HW/SW interfaces during high level specification of software applications because of the heterogeneity of emerging MPSoCs. According to this HW/SW interface point of view, in [228, 229] a video encoder (openDIVX, MPEG4) is parallelized using MPI on a MPSoC virtual platform.

In [230, 231] a message passing implementation based in producer-consumer communication is presented, taking in consideration the communication overhead and synchronization, but it does not use the standard MPI API, so the application code is not portable. Despite this fact, a large body of works have been presented on the design of MPI on top of MPSoCs. In [232, 233] a MPI-like microtask communication is presented upon Cell Broadband Engine™ processor [25], and on a small cluster on chip based using MicroBlaze and FSL channels, respectively. Nevertheless, both approaches have architectural scalability problems, since the maximum number of processing elements is eight on a Cell, and the maximum number of FSL channels is also eight in the MicroBlaze MPSoC architecture.

In [234] a SoC-MPI library is implemented on Xilinx Virtex FPGA, exploring different mappings upon several generic topologies. In [235] a custom MPI called rMPI have been reported and targeted to embedded systems using MIT Raw processor [236] and on-chip network. In this work, a traditional MPI implementation is compared with the rMPI, the developed version ported to embedded systems. Another interesting work, presented in [237], focuses on parallel programming of Multi-core Multi-FPGA systems based on message passing. This work presents TMD-MPI as subset of MPI, and the definition and extension of the packet format to communicate systems in different FPGAs, and intra-FPGA using a simple NoC architecture.

Some of these MPI-based implementations presented before [230, 232–235, 237] can be executed as stand alone without relying on an existing operating systems. Furthermore, each MPI library is implemented to have a very small footprint due to the memory-constraints of embedded and on-chip systems.

Despite this fact, a lightweight MPI for embedded heterogeneous systems (LMPI) is reported in [238] relying on the existence of an OS. However, this approach is more focused on a custom implementation of MPI for a traditional cluster of workstations.

Very recently, in [239, 240] a MPI task adaptive task migration message passing is explored using a HS-Scale MPSoC framework [241]. In this work, a homogeneous NoC-based MPSoC is developed for exploring scalable and adaptive on-line continuous mapping techniques using MPI. Each processor of this system is compact and runs a tiny preemptive operating system that monitors various metrics and is entitled to take remapping decisions through code migration techniques.

In addition, out of the academia research, the Multicore association proposed MCAPI [242]. This API is intended to join all previous work. It is targeted to multicore systems based on message passing, shared memory, but also it includes APIs to support load balancing, power and task management, as well as QoS and debug facilities.

In this dissertation, we focus on embedding MPI towards NoC-based systems. We believe that MPI is a good candidate to be the parallel programming model to map upon highly parallel and scalable NoC-based MPSoCs. The main reasons are:

- The inherent distributed and scalable nature of the communication backbone, the heterogeneity of computational elements in non-cache coherence NoC-based MPSoCs make message passing a priori the best model.

- MPI-based communication model fits perfectly on high scalable distributed multiprocessor systems with many cores interconnected by a on-chip network backbone, even on a single chip. A priori, shared-memory architectures using the OpenMP model are poorly scalable because of the limited bandwidth of the memory.

- The end-to-end low-latency interconnects of the elements allow fast inter-process communication using message passing (OpenMP often suffers due to the overhead produced by cache-coherence protocols).

- Explicit parallel programming is often hard but very useful to write efficient custom programs since NoC-based MPSoCs often are application-specific. To mitigate this, communication assistants, and automatic parallel code generators or compiler can be used.

- The portability and extensibility of MPI API make it easy to be tailored to NoC-based MPSoCs.

- Is a very well-know API and parallel programming model, and many debug, and trace tools are ready to be used.

- The reusability of parallel source codes used previously in HPC on embedded scenarios. This avoids the replication of efforts involved in developing and optimizing the code specific to the parallel template.

- Hides hardware complexities (NoC design space is huge) increasing programmability from the programmer point of view. Nevertheless, extensions can be included to enable low-level performance features (e.g. load balancing, latency management, power management, etc).

Furthermore, message passing has proven to be a successful parallel programming model for distributed memory machines, such as the grid infrastructure, supercomputers, clusters of workstations, and now due to the advantages about low-latency inter-processor communication and thanks to the inherent distributed nature of NoC-based MPSoCs the author believe that is the most suitable well-know parallel programming model.

Our contribution is the formal definition on how to embed standard MPI target to distributed-memory NoC-based MPSoCs taking into account all hardware and software issues. Thus, in this work we propose an efficient and lightweight implementation of an on-chip message passing interface, called on-chip MPI (ocMPI). This implementation does not rely, as some previous work, in any OS [238, 243], but it includes very efficient HW-SW interfaces using specific tight-coupled NIs, feature not present in any previous work. The outcome is a high-performance low-latency message passing injection rate of packets/messages on the NoC-based system.

Even if in this chapter we mainly focus in message passing, we also contributed on adding the QoS features presented in Chapter 4 on page 85, on top of an existing OpenMP framework specially targeted to MPSoCs. Therefore, applications and/or the programming environment chosen for their development/execution should have some degree of control over the available NoC services. In this way, the application programmer or the software execution support layer (either an OS, a custom support middleware

or runtime environment) can exploit the HW resources so as to meet critical requirements of the application and, more in general, speed-up threads and the execution while fitting a limited power budget.

On both scenarios and parallel programming models, message-passing or shared-memory NoC-based MPSoC, hardware architectures and software libraries have been tuned to get high-performance inter-processor communication. This fact, is not explicitly present in any previous work.

6.5 Embedding MPI for NoC-based MPSoCs

This section describes all the features and key points faced during the adaptation of the standard Message Passing Interface (MPI) towards on-chip architectures. Thus, we present the development of two suitable software stacks based on MPI [202] to be used as the core of message passing parallel programming model upon different NoC-based MPSoC architectures:

- on-chip Message Passing Interface (ocMPI): fully developed in ANSI C (to avoid large memory footprints and in order to achieve a maximum runtime performance), which can be compiled in many real MPSoC architectures which integrates processors (e.g. ARM, Nios II, MicroBlaze, PowerPC, DSPs, etc) that support gcc-like tool chain.

- java Embedded Message Passing Interface (j2eMPI): written completely in Java in order to be the parallel programming model upon NoCMaker, our open source NoC-based MPSoC cycle accurate simulator [51, 244] presented in Section 2.4 on page 36.

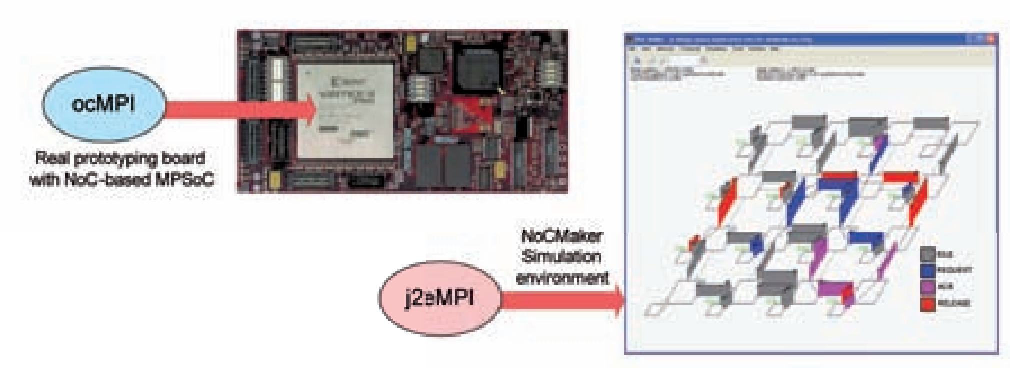


Figure 6.2: On-chip message passing interface libraries

Both software stacks and APIs are completely equivalent with the particularity that ocMPI is designed and targeted to be used in a memory-constrained NoC-based MPSoCs in real prototyping platforms, whereas j2eMPI has been implemented to run on top of NoCMaker.

The advantage to have both message passing APIs (i.e. ocMPI and j2eMPI) for NoC-based MPSoC, let us to run simulation and applications on NoCMaker, but also to execute them in a fast way on a real prototyping platform. Finally, comparing the results of both executions, we can tune and increase the accuracy of our simulator according to the information extracted from the real platform.

6.5.1 Message Passing Interface (MPI) - General Concepts

MPI is a portable [245] C/C++ standard software API to create and communicate parallel distributed-memory applications. MPI-2 software API contains over 125 routines to perform distributed parallelism, but programs are essentially based on two basic routines to perform a peer-to-peer communication, `MPI_Send()` and `MPI_Recv()`. MPI-2 is the second version of MPI that extends the functionality of MPI-1 adding parallel I/O support, adding bindings to C++ and being more portable to different platforms.

As shown in Figure 6.3, MPI is based on the communicator concept that is an ordered set of processes indexed by rank that remains constant from its creation until its destruction. Inside it N processes can live being identified with a rank range between *0* to *N-1*; the whole set of MPI processes form the `MPI_COMM_WORLD` communicator. In addition, MPI users are able to define their own communicator which will be a subset of `MPI_COMM_WORLD`.

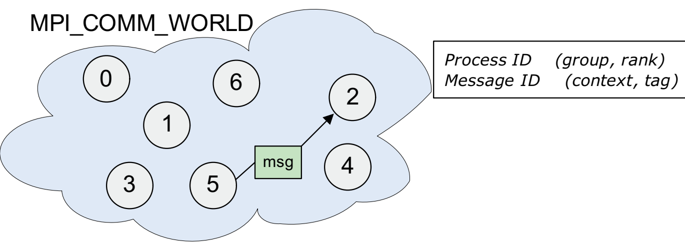


Figure 6.3: Logical view of communicators and messages in MPI

In MPI, a process involved in a communication operation is identified by group and rank within the group, whereas, messages may be considered labeled by communication context and message tag within that context. Thus, through `MPI_Comm_Size()` and `MPI_Comm_rank()` functions, we are able

to answer two important questions in parallelism, how many processes are involve? and who am I?, respectively.

MPI contains a large set of *point-to-point communication routines* between two processes. There are available some ways to send data:

- Blocking mode - `MPI_Send()`: the function does not complete until all message is sent and received using `MPI_Recv()`. In this mode, the completion depends on message size and amount of system buffering.

- Synchronization mode - `MPI_Ssend()`: the sender notifies the receiver; after the matching receive is posted the receiver acknowledges back and the sender sends the message, so the function does not complete until a matching receive has begun (e.g. like a fax). Unsafe programs become incorrect and usually deadlock within `MPI_Ssend()`.

- Buffered mode (asynchronous) - `MPI_Bsend()`: the user supplies the buffer to the system for its use. If there is not enough space, the message is saved in a temporary buffer until someone retrieves it (e.g. like e-mail). In other words, the sender notifies the receiver and the message is either buffered on the sender side or the receiver side according to size until a matching receive forces a network transfer or a local copy respectively. Users must supply enough memory to make unsafe program safe.

- Ready mode - `MPI_Rsend()`: similar than synchronous but without acknowledging that there is a matching receive at the other end. Allow access to faster protocols but has undefined behaviour if the matching receive is not present.

Note that there exists non-blocking modes of these sending modes, `MPI_Isend()`, `MPI_Issend()`, `MPI_Irsend()`, `MPI_Ibsend()`. It is important to remark that, all these non-blocking functions return immediately, but can be waited or queried, too. In addition, all types of sending functions or modes may be received with `MPI_Recv()`.

It also include a great variety of *collective communication routines* to communicate processes in a group. Basically these communication routines can be classified as:

- one-to-many (e.g. `MPI_Broadcast`, `MPI_Scatter()`,...).

- many-to-one (e.g. `MPI_Gather()`, `MPI_Reduce()`,...).

- many-to-many (e.g. `MPI_Barrier()`, `MPI_AlltoAll()`,...).

In Figure 6.4, we show the typical traffic patterns of these collective communication routines.

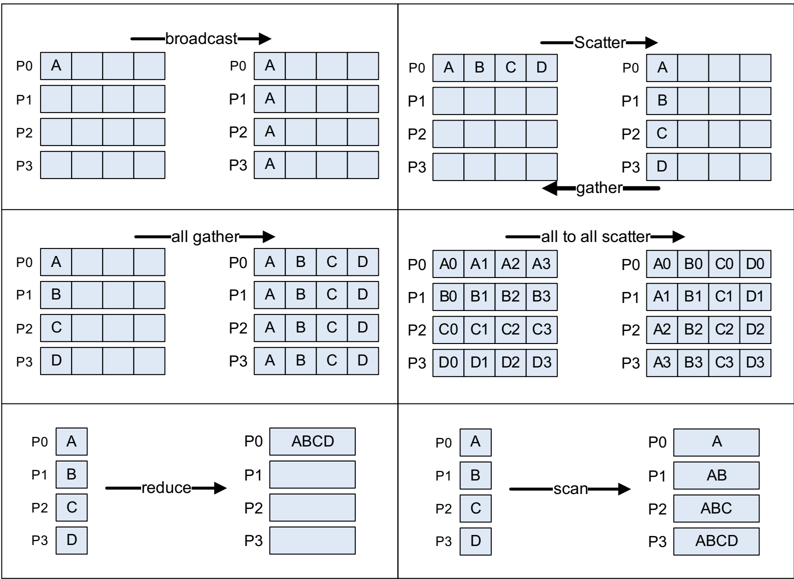


Figure 6.4: Patterns of the MPI collective communication routines

MPI-2 defines basic data types, but also the possibility to define custom data types in order to get two purposes, heterogeneity between different processors, and work with non-contiguous data types (e.g. structures or vectors with non-unit stride). Thus, the MPI data types could be classified as:

- Basic or elementary: language-defined data types (e.g. `MPI_INT`, `MPI_DOUBLE`, `MPI_BYTE`, `MPI_UNSIGNED_INT`,...).
- Vector/Hvector: array separated by constant strides (e.g. an array of integers, or chars). In Hvector the stride is defined as a byte.
- Indexed/Hindexed: array of indices (used for scatter and gather). In Hindexed is indexed with indices in bytes.
- Struct: general used in mixed types (e.g. heterogeneous structures in C).

Of course, MPI has more complex functionality that the outlined above, for example, it contains routines concerning IO parallel functions, creation of virtual topologies, etc. Nevertheless, in most cases when we use MPI to develop parallel programs, only a subset of the MPI primitives are used. In

addition, all MPI features explained before are the most important from the architectural and communication point of view to make easy to understand this chapter.

6.5.2 MPI Adaptation Towards NoC-based MPSoCs

In this section, we explain the software design to create our message passing communication software stack and its associated API library for NoC-based systems. In this work two main phases have been performed to port the standard MPI towards NoC-based systems by using a bottom-up approach. This is possible because MPI is highly portable parallel programming model [245].

- We selected a minimal working subset of standard MPI functions. This task is needed because MPI contains more than one hundred functions, most of them not useful in NoC-based MPSoC scenarios (e.g. generation of virtual topologies).
- We make an accurate porting process (see Figure 6.5) by modifying the lower layers of each selected function of standard MPI in order to send/receive MPI packet by creating transactions through the NI to the NoC, and vice versa.

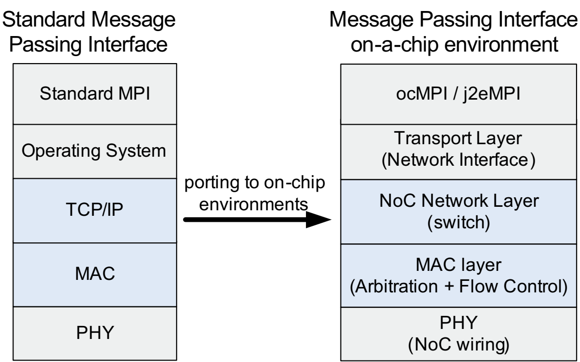


Figure 6.5: MPI adaptation for NoC-based systems

The on-chip MPI software stack presented in Figure 6.5, hides NoC-level features from high-level parallel programming model. In other words, from the point of view of the software, the API library to perform message passing is completely independent from NoC architectural choices, such as topology, flit width, arbitration policy, switching communication scheme, etc.

6.5.3 Encapsulation, and on-chip MPI Packet Format

This ocMPI message is divided in two parts, a header of 20 bytes with contains the message passing protocol information, and a variable length payload which essentially contains the payload data to be sent. As shown in Figure 6.6, each ocMPI message has the following envelope: *(i)* Source rank (4 bytes), *(ii)* Destination rank (4 bytes), *(iii)* Message tag (4 bytes), *(iv)* Packet datatype (4 bytes), *(v)* Payload length (4 bytes), and finally *(vi)* The payload data (a variable number of bytes).

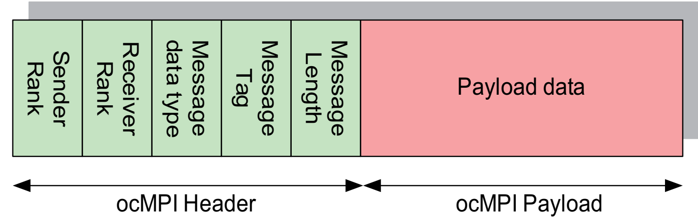


Figure 6.6: onMPI/j2eMPI message layout

Later, this ocMPI high level message will split in packets/transactions or stream of flits according to the channel width of the communication infrastructure, in our case a NoC.

6.5.4 A Lightweight on-chip MPI Microtask Stack

Due to the on-chip memory constrains that we have on MPSoCs architectures, and in order to do an efficient and lightweight version of MPI on top of NoC-based systems, we started the porting process from scratch based on the source code of the Open MPI-2 initiative [204]. In addition, we carried out the porting process by using an incremental bottom-up approach with several refinement phases following the methodology presented in [243].

Mainly, in order to design our lightweight ocMPI we follow the next considerations:

- All MPI data structures have been simplified in order to get a reduced memory footprint. The main structures that have been changed are the communicator, the datatypes and the operations. Thus, we only create the default communicator, `MPI_COMM_WORLD`, and its associated group.
- No user-defined datatypes and operations are supported. Only the basic MPI datatypes and operations are defined (e.g. `MPI_INT`, `MPI_SHORT`, `MPI_UNSIGNED_SHORT`, etc.).

- No support for virtual topologies have been added because application-specific or custom NoC designs can be modeled by using the tools presented in [50,51,68,124].
- No Fortran bindings and MPI I/O functions have been included.
- All ocMPI/j2eMPI functions follow the standardized definition and prototype of MPI 2.0 to keep this portability.

Table 6.2 shows the 20 standard MPI functions ported to our NoC-based MPSoC platform.

Types of MPI functions	Ported MPI functions
Management	`ocMPI_Init()`, `ocMPI_Finalize()`, `ocMPI_Initialized()`, `ocMPI_Finalized()`, `ocMPI_Comm_size()`, `ocMPI_Comm_rank()`, `ocMPI_Get_processor_name()`, `ocMPI_Get_version()`
Profiling	`ocMPI_Wtick()`, `ocMPI_Wtime()`
Point-to-point Communication	`ocMPI_Send()`, `ocMPI_Recv()`, `ocMPI_SendRecv()`
Advanced & Collective Communication	`ocMPI_Broadcast()`, `ocMPI_Barrier()` `ocMPI_Gather()`, `ocMPI_Scatter()`, `ocMPI_Reduce()`, `ocMPI_Scan()`, `ocMPI_Exscan()`

Table 6.2: Supported functions in our on-chip MPI libraries

This ocMPI implementations is completely layered and advanced communication routines (such as `ocMPI_Gather`, `ocMPI_Scatter()`, `ocMPI_Bcast()`, etc.) are implemented using simple point-to-point routines, such as `ocMPI_Send()` and `ocMPI_Receive()`.

Furthermore, our ocMPI implementation does not depend of an OS, and in consequence, there is not any daemon, such as *mpd* running allowing to launch *mpirun/mpiexec*. Thus, we define the number of processors involved in the NoC using a global variable in a configuration file. Then, we specify who is the master at compile time by using a pre-compiler directive `-DMASTER` on the processor that we require to act as a master of the NoC-based MPSoC. Finally, the assignment of the rank to each processor is performed at runtime when we call `ocMPI_Init()` function. This enables the possibility to create different application mappings depending on the application requirements. The definition of the cores involved in the NoC-based MPSoCs is set by a constant named `ocMPI_CPUs`[2] declared also in a con-

[2]This constant expresses the number of available processors involved in the NoC-based MPSoC, which can be different from the actual available processors on the platform.

figuration file. This is equivalent to `-np` option on `mpirun/mpiexec` command. Later, as usual, we can obtain how many processors are involved in the communication and also to know their rank using `ocMPI_Comm_size()` and `ocMPI_Comm_rank()` functions, respectively.

6.5.5 Software Stack Configurations

Depending on the application requirements, we can select different software configuration stacks. Figure 6.7 lists four typical configurations of the ocMPI library and its stripped code size in bytes[3]. The minimal working subset of our ocMPI requires only 4.942 bytes of memory, and it contains the NI driver and middleware routines and the following six functions:

- `ocMPI_Init()` & `ocMPI_Finalize()`: initializes/ends the ocMPI software library.
- `ocMPI_Comm_size()`: gets the number of all concurrent processes that run over the NoC-based MPSoC
- `ocMPI_Comm_rank()`: gets the rank of a process in the ocMPI software library.
- `ocMPI_Send()` & `ocMPI_Recv()`: blocking send/receive functions.

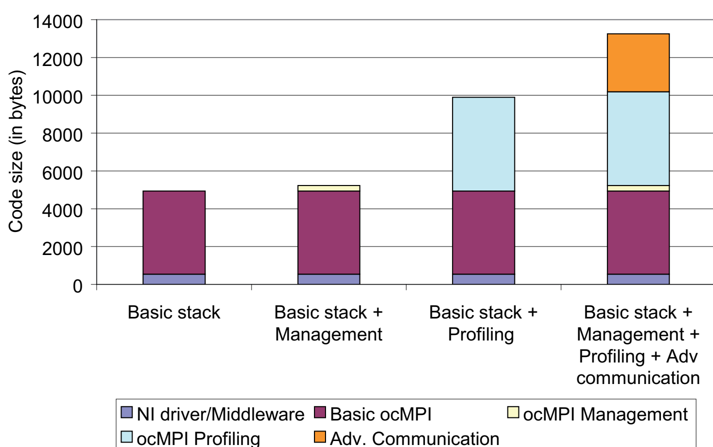


Figure 6.7: Different software stack configurations (ocMPI library)

[3]The results shown in Figure 6.7 have been extracted using the `nios2-elf-gcc` compiler with `-O2` code size optimization.

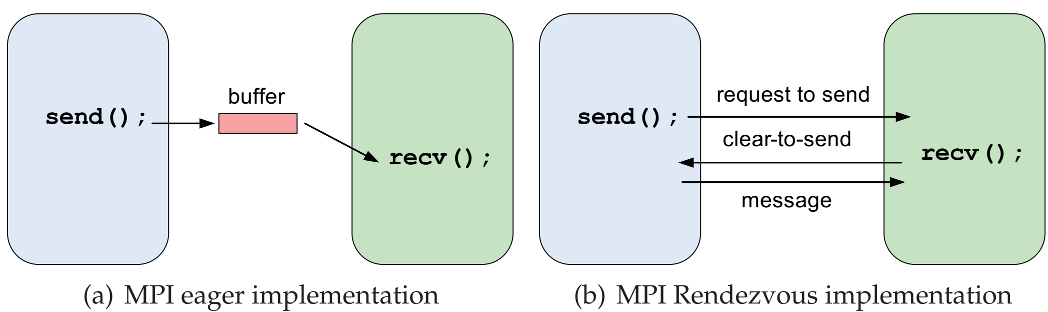
Other interesting software configuration is the basic stack plus the ocMPI profiling functions. As shown in Figure 6.7 on the previous page, these profiling functions employ large code size because must include both, many Nios II HAL API functions calls, and libraries to access the required peripherals (i.e. the timers and/or the performance counter). This is an important point to be optimized in the future by creating custom APIs and peripherals.

Finally, Figure 6.7 on the preceding page shows that, the complete ocMPI library with the advanced collective communication primitives only takes 13.258 bytes, which makes it suitable for embedded and on-chip systems.

6.5.6 Message Passing Communication Protocols

Message passing protocols (see Figure 6.8), as well as intermediate buffers in MPI determines different trade-offs at the moment of pack/unpack short and long messages in the runtime library (in conjunction with the hardware architecture). These elements determine the final performance of our interprocessor on-chip MPI communication library.

Most of the MPI implementations employ a two-level protocol for point-to-point messages. Short messages are sent eagerly (see Figure 6.8(a)) to optimize the communication in terms of latency, while long messages are typically implemented using a rendezvous mechanism (see Figure 6.8(b)).



(a) MPI eager implementation (b) MPI Rendezvous implementation

Figure 6.8: Eager vs. rendezvous message passing communication

The rendezvous implementation is fully synchronous. The sender must first send a request which includes only the message envelope with the details of the MPI message (basically the MPI message header presented in Figure 6.6 on page 151). Later, if the header matches with the receiver, it will reply with an acknowledgment (often called as clear-to-send), and finally, the sender will transmit the actual data to be transferred. Furthermore, depending if the `send()` arrives before the matching `recv()`, or the opposite, the sender or the receiver process must be suspended, respec-

tively. This fact can lead to early sender, late receiver performance inefficiencies.

On the other side, an ”eager” implementation allows a send operation to complete without the corresponding receive operation is executed. This protocol assumes enough buffering capabilities on the system, and often it is translated to synchronous and buffer communication schemes. This happen when a send stalls because all available buffer space is full, then the send routine will wait until buffer become re-available.

Rendezvous has higher message delivery overhead due to the synchronization process than eager protocol. However, since the message to be sent is large, the overhead of the communication protocol with the receiver is assumed to be amortized by the transfer. Under rendezvous, the required buffer space is to hold MPI header (20 bytes for ocMPI).

In our ocMPI implementation, we assumed not very long MPI messages (few KBs), due to the memory constraints, and the reduce data sets and workloads used on NoC-based MPSoCs. Therefore, we believe that an eager protocol, or potential variants to improve under long messages [246], will outperform rendezvous protocol, which has been implemented in [237].

In addition, it is obvious that local memories in processor nodes can not be as large in traditional distributed memory multiprocessors. Instead, scratchpads memories of reasonable size can be used, since they exhibit a negligible access/energy cost.

In any case, further research will be desirable in order to compare more quantitatively both message passing communication protocols.

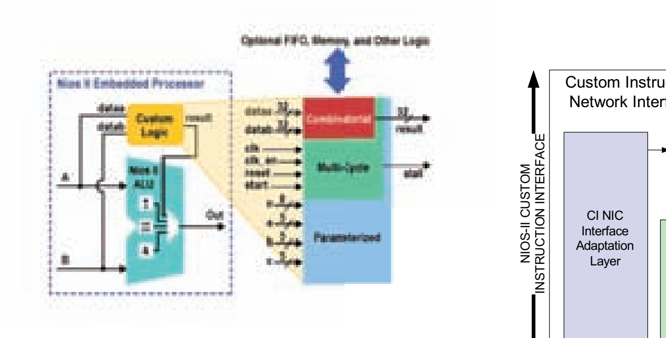
6.5.7 Improving ocMPI Through Tightly-Coupled NIs

The shortcoming of previous work [232–235, 237], is probably the lack of analysis and improvement between hardware and software interfaces, which it is present in [230, 247], but definitely it is still now an open issue.

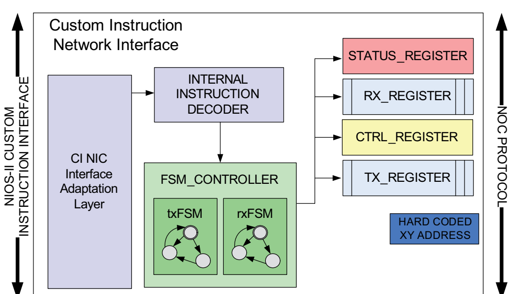
In this work, we introduce an exotic tight-coupled NI implemented as an assembly custom instruction (CI) inside the processors datapath. This NI implementation is quite different from the previous ones. In this solution, we exploit the flexibility offered by the soft-core Nios II architecture to integrate our own CIs in the ALU within the processor datapath (see Figure 6.9(a) on the next page). This feature lets us to extend the basic Nios-II instruction-set by adding our NoC-based tight coupled custom instructions, as well as it allows to pipeline messages, when they are sent and received.

Obviously, in this implementation, it behaves like a functional unit or as a co-processor to boost the message passing. The interface of a custom instruction on a Nios II as shown in Figure 6.9(a) on the following page is

essentially based on two input operands `data[31:0]` and `datab[31:0]`, used to pack the data and the destination address; and the signal `n[7:0]` used to select an specific operation, i.e. `ciNiSend()`, `ciNiRecv()` or `ciNiStatus()`. Finally, the `start` signal that is asserted by the processor when the CI starts. The CI generates two outputs, `result[31:0]` containing data under read operations, and `done` indicating when the output is valid and available.



(a) Custom instruction on Nios II



(b) Tight-coupled NI block diagram

Figure 6.9: Tight-coupled NI block diagram

In contrast to bus-based NI implementations, the tight-coupled NI is not memory mapped on the addressable memory space. Instead, it is attached directly through a CI inside the CPU, and it has a set of status, control and data registers as shown in Figure 6.9(b).

- 32-bit Status Register (0x0).
- 32-bit Receive Register (0x4) – Configurable as a FIFO.
- 32-bit Control Register (0x8).
- 32-bit Transmitting Register (0xC) – Configurable as a FIFO.

The status register includes different status bits in order to identify whether the NI is busy, the queue is full, or there is data to be read. TX and RX are the registers to send and receive data, whereas the control register is included for advanced future usage. Based on the address the tight-coupled NI is able to send/receive messages from any IP core on the NoC-based MPSoC. The behaviour of tight-coupled NI relies on the producer/consumer synchronization.

Next, in Listing 6.1 are show the three main custom instructions implemented.

Listing 6.1: Tight-coupled NI instructions

```
// Check NI status register.
// Return the status as a 32-bit integer.
inline int ciNiStatus(void);

// Send 32-bits data to a specific address.
// Return 1 for success, otherwise return < 0.
inline int ciNiRecv(int *data, int address);

// Receive 32-bits data from an specific address on buffer.
// Return 1 for success, otherwise return < 0.
inline int ciNiSend(int *data, int address);
```

Using this `ciNiSend()` and `ciNiRecv()` other specific data exchanging routines according to determined MPI datatypes can be defined.

```
inline int ciNicSend(byte *buffer, int length, int address);
inline int ciNicRecv(byte *buffer, int length, int address);
inline int ciNicSendChar(char caracter, int address);
inline int ciNicSendDouble(double data, int address);.....
```

As shown in Section 6.7.2 on page 160, under point-to-point unidirectional or bidirectional traffic our tight-coupled NI can reach better injection rates than a simple NI which is attached to the system-bus. This is because, we remove arbitration and decoding phase of buses like AMBA AHB. Nevertheless, this type of exotic NI can be only applicable to soft-core processors, where the hardware designer can tune the processor datapath.

6.6 OpenMP Framework for NoC-based MPSoCs

In this section we introduce our contribution on the OpenMP framework reported in [183] by adding all QoS features presented in Chapter 4 on page 85 on top of the runtime of the parallel programming model or within the OpenMP application. This fact will enable QoS features present at NoC-level from the software application level.

The OpenMP implementation consists of a code translator and a runtime support library. The framework presented in this paper is based on the GCC 4.3.2 compiler, and its OpenMP translator (GOMP). The OMP pragma processing code in the parser and the lowering passes have been customized to match the peculiarities of our architecture. All the major directives and clauses are supported by our implementation. Work-sharing

constructs support all kinds of static and dynamic scheduling, and all synchronization constructs are supported. The runtime library (libgomp) has been re-implemented from scratch due to the absence of an homogeneous threading model.

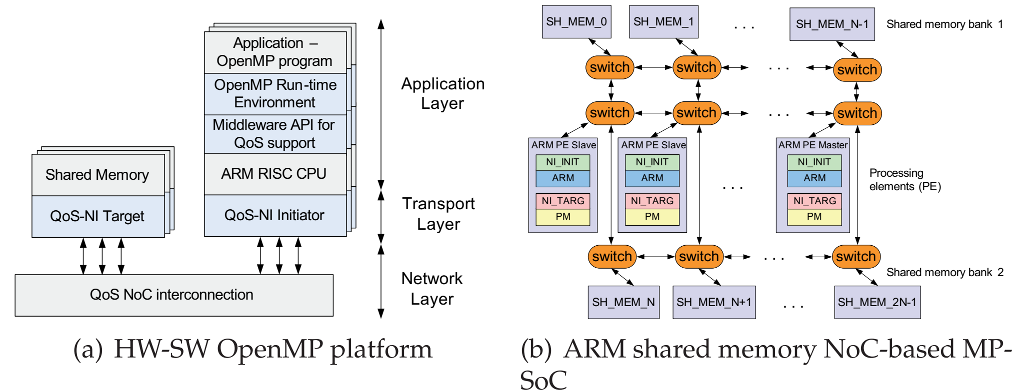
The original implementation of the *libgomp* library is designed as a wrapper around the pthreads library. A `#pragma omp parallel` directive is handled outlining the code within its scope into a new function that will be mapped to parallel threads through a call to `pthread_create`. In our case, we need to take a different approach, since we cannot rely on a thread library that allows us to fork a thread onto another core.

The runtime library implements the `main()` function. Each processor loads its executable image of the program, and then the execution is differentiated between master and slave threads based on the core ids. After an initialization step, executed by every thread, the master calls the application `main()` function while the slaves wait on a barrier.

The custom translator replaces every `#pragma omp parallel` directive with a call to our library function `do_parallel()`, and passes it two arguments: the address of the parallel function code, and the address of a memory location where shared data is placed. In the function `do_parallel()`, the master updates the addresses at which the slaves will look for the parallel function and the shared data. At this point it releases the barrier and all the threads can execute concurrently the outlined function. At the end of the parallel region the master core continues executing sequential parts of the application, while the slaves come back on the barrier.

In the OpenMP programming model barriers are often implied at the end of parallel regions or work-sharing directives. For this reason they are likely to overwhelm the benefits of parallelization if they are not carefully designed taking into account hardware peculiarities and potential bottlenecks. This may happen when the application shows load imbalance in a parallel region, or when most of the execution time is spent in serial regions. The latter happens because the remote polling activity of the slaves on the barrier fills the interconnect with access requests to the shared memory and to the semaphore device. To address this issue we devised a barrier implementation that exploits the scratchpad memories. The master core keeps an array of flags representing the status of each slave (entered or not), and notifies on each slave's local memory when everybody has arrived on the barrier. Moving the polling activity from the shared memory to local scratchpad memory and eliminating the necessity for lock acquisition allows us to remove the congestion on the interconnect.

As shown in Figure 6.10, the embedded OpenMP parallel programming model have been mapped on top of a distributed shared memory NoC-based MPSoC virtual architecture developed using MPARM [54] and ×pipes [52,53,69]. In Figure 6.10(a), we depict a general hardware template of our parallel architecture and the OpenMP software components.



(a) HW-SW OpenMP platform

(b) ARM shared memory NoC-based MPSoC

Figure 6.10: HW-SW view of our OpenMP platform

Our architecture template, as shown in Figure 6.10(b), is a regular 2D-Mesh NoC-based MPSoC platform, based on a NoC mesh, which consists of a configurable number of ARM processors and shared memories.

The system contains a master processor (ARM PE master) which is in charge of supervising the execution, either dynamically or statically, setting up or tearing down the NoC QoS features through the middleware functions (see Section 4.4 on page 96), of distributing and mapping data around the shared memories on the OpenMP application. Normally this is the only processor which will reconfigure the QoS features using the profiled OpenMP application, but it is important to remark, that the middleware routines can be invoked by any other slave processor during the execution of the application if useful.

On the other hand, the system also contains a set of slave processors (ARM PE slave) which are in charge of execute different parts of OpenMP kernels distributed by the master. In addition to the PEs, the system includes two banks of shared memories. The first bank is used for the OpenMP library runtime support, while the second bank is used to store the shared data of the OpenMP application. Obviously, the NoC fabric, as well as the ARM PEs and the shared memories include all QoS features presented in Section 4.3 on page 89, including QoS NIs initiators/targets and QoS switches, respectively.

6.7 Evaluation of on-chip Message Passing Framework

This section shows the results extracted from the different parallel programming models, i.e. ocMPI/j2eMPI using architectures created by NoC-Maker [51] on Nios-II prototyping platforms, and the OpenMP runtime framework on an ARM NoC-based MPSoC (see Figure 6.10(b) on the preceding page).

6.7.1 Profiling the Management ocMPI Functions

This section presents the profiling of ocMPI management functions to demonstrate its minimum execution overhead. Figure 6.11 presents the results acquired by performing a monitoring with external non-ocMPI profiling function during the execution of each parallel program. The results shown that our ocMPI management functions consume around 75 cycles, with the exception of `ocMPI_Get_processor_name()` function spends around 210 clock cycles.

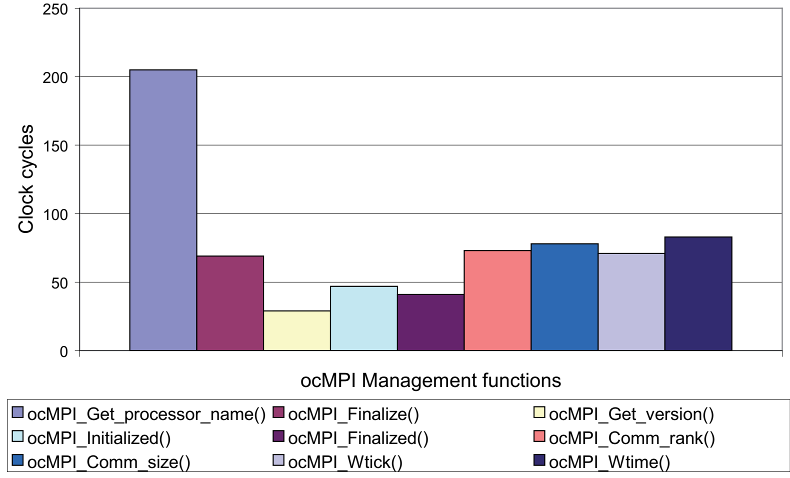


Figure 6.11: Profiling of the management ocMPI functions

6.7.2 Low-level Message Passing Benchmarks

This section shows the evaluation and profiling of all software layers of ocMPI/j2eMPI API for message passing. Thus, in this section we

characterize our ocMPI library by executing several benchmarks in order to extract the most important communication metrics. Performance parameters such as link latency, link bandwidth, and synchronization cost are measured by executing a set of customized micro-benchmarks [248,249].

The process to execute the different micro-benchmarks is the following:

(1) Start the timer – `ocMPI_Wtime()`.

(2) Loop modifying the parameters of the micro-benchmark (e.g. size message by power of 2, number of executions, etc).

(3) Verify if the micro-benchmark finishes and the system is stable.

(4) Stop the timer – `ocMPI_Wtime()`.

(5) Compute the metric.

Firstly, to evaluate the real bandwidth and the injection rate from low-level functions on the NI, we performed some basic tests, like unidirectional or ping-pong tests.

Looking at the results on Figure 6.12, the use of the tight coupled NI and the low-level functions routines is the best choice to communicate any Nios II based tiles in our NoC-based MPSoC[4].

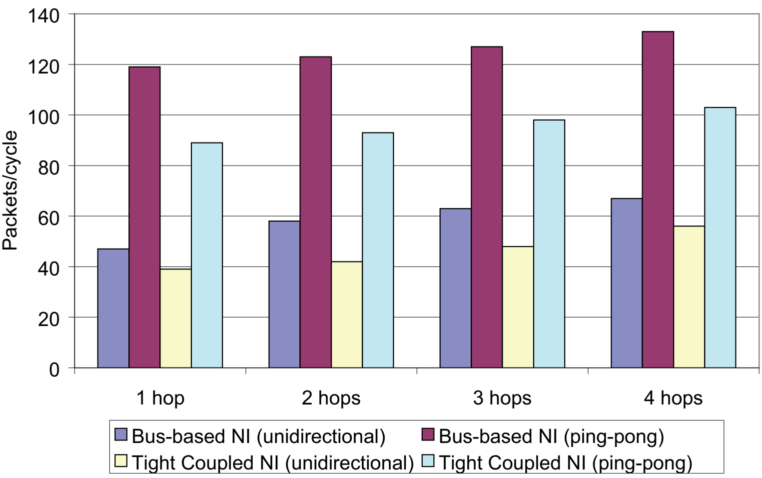


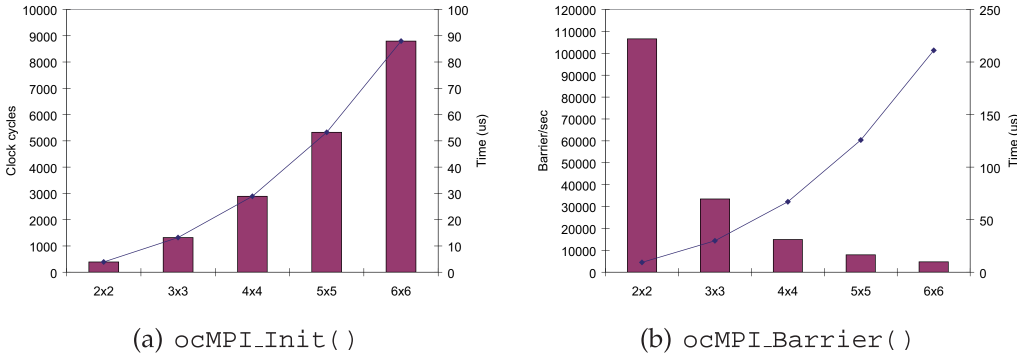
Figure 6.12: Comparison of a bus-based vs. tight coupled NI

[4]We used a standard Nios II soft-core processor, on top of an Ephemeral circuit switching NoC-based system

The results show that our tight coupled NI integrated in the processor data-path is faster than our bus-based NI. It offers around 80 Mbps (counting the software stack) of bandwidth under unidirectional traffic patterns, and the injection rate is close to 1 packet every 40 cycles at only 100 MHz. Figure 6.12 on the previous page shows injection rates for both NI implementations for our Ephemeral circuit switching. Obviously, in all cases the bandwidth and the injection rate is degraded according to the number of hops due to end-to-end flow control protocol on the NoC. However, this degradation is only slightly increasing with the number of hops, and therefore it permits to scale up to large N-dimensional mesh NoCs.

Second, we execute some ocMPI functions to profile them by extracting the execution time. Scalability tests have been done by increasing the number of IP cores. The `ocMPI_Init()` notifies all the processors on the NoC the rank and the involved nodes on the cluster on chip.

In Figure 6.13(a), we show the execution time in clock cycles (and the equivalent in us using 100 MHz clock) to initialize the NoC in different 2D-Mesh NoC-based MPSoC, where the root processor is placed on the left-bottom corner. The latency to set up the assign dynamically the ranks on each processor on the NoC ranges from 3,88 μs for quad-core until 87,96 μs on a 36-core MPSoC. These times are considered negligible compared to the execution time of the applications we will run, and furthermore, it is only done once when ocMPI stack starts.



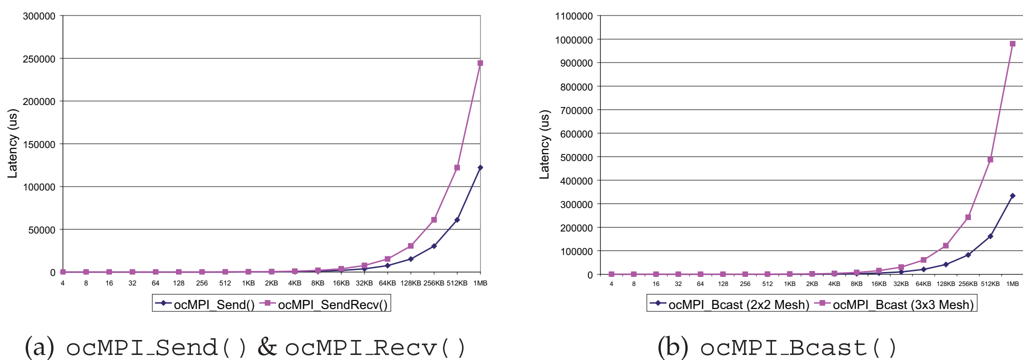
(a) `ocMPI_Init()` (b) `ocMPI_Barrier()`

Figure 6.13: Execution time of `ocMPI_Init()` & `ocMPI_Barrier()`

In message passing, usually barriers are required on MPI programs to synchronize all CPUs involved in a parallel task. In Figure 6.13(b), we report the evolution of the synchronization time in multiple 2D-Mesh NoC-based MPSoCs architectures. The plot shows the number of barriers/sec we can run on the MPSoC, as well as, the execution time in μs of a single barrier using a clock frequency of 100 MHz.

These metrics, while are topology dependent, shows the overall system performance and its degradation when the number of cores increase. Both, `ocMPI_Init()` and `ocMPI_Barrier()` have been implemented without using any optimization. Thus, the broadcast notification performed in `ocMPI_Init()` is based on sending *N-1* messages from root to the rest of slave processors, whereas `ocMPI_Barrier()` is implemented on a pure master-slave software implementation. Software barrier techniques reported in [250] can be a potential improvement by means of using tree or butterfly approaches.

Finally, in Figure 6.14(a) is plotted the execution time for `ocMPI_Send()` under unidirectional traffic on the NoC-based MPSoC. On the same figure, the time to execute `ocMPI_Send()` and `ocMPI_Recv()` is reported under bidirectional traffic. In both, benchmarks the size of the ocMPI payload have been change from values ranging from 4 bytes to 1 MByte. On message passing programs, a typical function to share data among the cores on the NoC is `ocMPI_Bcast()`. In Figure 6.14(b), we show the execution times to perform a broadcast in 2x2 and 3x3 NoC-based MPSoCs.



(a) `ocMPI_Send()` & `ocMPI_Recv()` (b) `ocMPI_Bcast()`

Figure 6.14: Execution time of point-to-point and broadcast in ocMPI

As expected, the execution time on a 3x3 Mesh increases exponentially in front of 2x2. The implementation of `ocMPI_Bcast()`, as well as the remaining collective communication routines, are implemented using point-to-point functions (i.e `ocMPI_Send()` and `ocMPI_Recv()`). Furthermore, its implementation use *N-1* point-to-point messages from the broadcast root to the other nodes inside the communicator. This behaviour can be improved, for instance, by means of using a row-column broadcast scheme. As before, the execution time is reported varying the packet size from 4 bytes to 1 MByte.

All these benchmarks are taking into account the overhead to execute the ocMPI software stack on top of the NoC-based MPSoC architecture based on

a Nios II processors. In addition, in our architectures, we never used a DMA to transfer data, so depending on the application mapping and communication patterns the computation not always is concurrent with the communication.

6.7.3 Design of Parallel Applications using ocMPI library

In order to verify our ocMPI/j2eMPI library a large variety of tests have been applied on different NoC-based MPSoC architectures, as well as high-level real application test sets involve well-known parallel applications.

Different parallel applications have been tested using our ocMPI/j2eMPI.

- Parallel calculation of π: this parallel version to compute π is based on the following summation approximation presented in Equation 6.1.

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \tag{6.1}$$

- Parallel generation of fractals: a fractal image is generated taking complex number representations and applying, for every point of a determined area, one simple recursive formula.
  The Mandelbrot set M is defined by a family of complex quadratic polynomials:

$$f_c : C \rightarrow C \quad givenby \quad f_c(z) = z^2 + c \tag{6.2}$$

  In Equation 6.2, c is a complex parameter of an arbitrary area. For each c, one considers the recursive behaviour of the sequence, $(0, f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), ...)$ obtained by iterating $f_c(z)$ starting at $z = 0$.

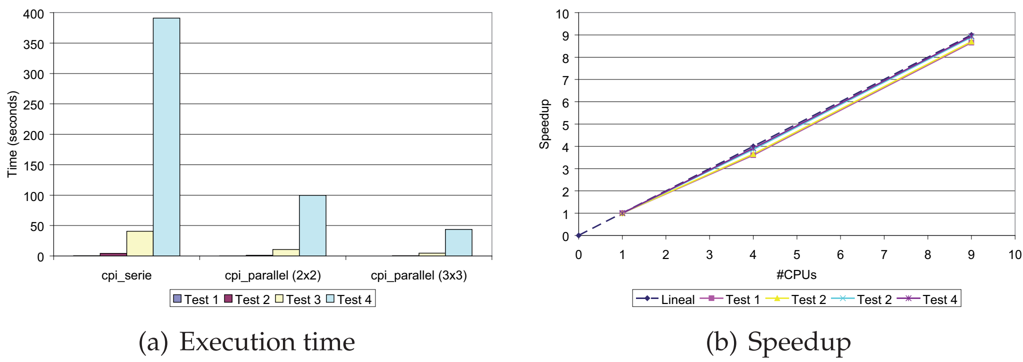
Both applications have been parallelized in a distributed master-slave mode through ocMPI on different multi-core architectures, such as a 2x2 and 3x3 Mesh architectures, with four and nine processors, respectively. Furthermore, it is important to remark that, the source codes of both applications have been compiled and executed without any relevant changes from the source code used in any traditional large-scale distributed system.

As shown in Table 6.3 on the facing page, for each parallel application, we try four different tests by changing the problem size parameters (i.e. N as pi precision, and MAX_ITERATIONS to fix the definition of the Mandelbrot set using a fixed 176x144 QCIF image).

Test #	Problem size
Test 1	N=10000 MAX_ITERATIONS=64
Test 2	N=100000, MAX_ITERATIONS=128
Test 3	N=1000000, MAX_ITERATIONS=256
Test 4	N=10000000, MAX_ITERATIONS=512

Table 6.3: cpi and Mandelbrot set problem size parameters

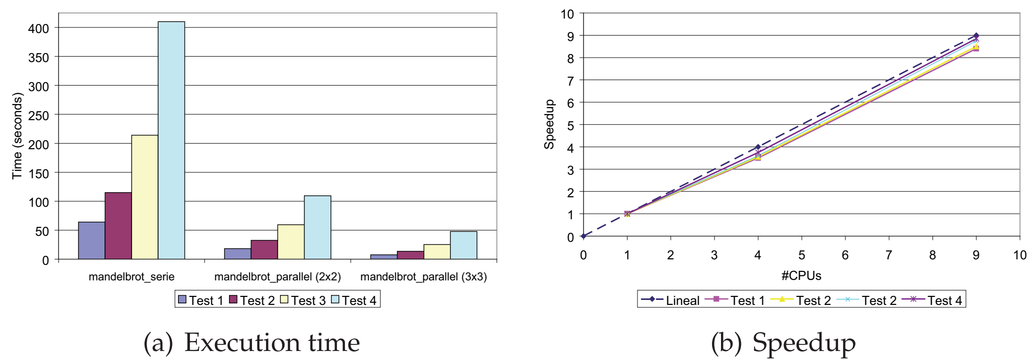
Figure 6.15(a) shows the execution time of cpi application according to the executing of each test obtained by `ocMPI_Wtime()`. In Figure 6.15(b), we show the speedup according to the number of processors by changing the pi precision. It is easy to observe that using parallel implementations we can achieve speedups close to the ideal case (blue discontinuous line), i.e. according to the number of CPUs integrated in our parallel embedded platform. For instance, the serial generation of π on *Test 4* takes 390,2 s, whereas its parallel version takes 99,71 s and 43,6 s, on our NoC-based MPSoCs with four and nine processors, respectively.



(a) Execution time (b) Speedup

Figure 6.15: Parallelization of cpi computation

On the other side, in Figure 6.16 on the next page, is presented the results of the parallel and serie generation of Mandelbrot set. Figure 6.16(a) on the following page) show the execution times by changing the architecture and the number of maximum iterations (see for each test, and the speedup of the parallel generation of Mandelbrot set. Thus, while *Test 4* takes around 410 s in serie, whereas the parallel generation takes 109 s or 47 s, respectively, on our 2x2 and 3x3 MPSoC Mesh architecture. As we can see in Figure 6.16(b) on the next page, the speedups are again very close to the ideal (blue discontinuous line). This is because each point of the Mandelbrot can be calculated independently and there are no data dependencies.



(a) Execution time (b) Speedup

Figure 6.16: Parallelization of Mandelbrot set application

Because both applications have been parallelized using a large granularity, so the computation-communication ratio is high, and due to the independence of the data set, both parallel applications show a very high scalability. In addition, it is important to remark that, the on-chip network communication latencies are almost negligible versus the computation time on each processor, and as a consequence, in both experiments we get speedups close to the number of processors.

6.8 Improve OpenMP Kernels using QoS at NoC-level

OpenMP is a widely adopted shared memory programming API. It allows to incrementally specify parallelism in a sequential C (or C++, or Fortran) code through the insertion of specific compiler directives. This work is based on the OpenMP implementation presented in [183], on top of which, we integrated our extensions to support QoS-related features.

There are several ways in which priorities and establishment can be exploited within the OpenMP programming framework. One possibility is to extend the API with custom directives to trigger prioritized execution for a particular thread (or group of threads). This option is useful to give the knowledgeable programmer the possibility of specifying appropriate prioritization patterns at different program points as needed. On the other hand, keeping the programmer in the decision loop requires insights on both program behavior and architectural details, thus compromising ease of programming. A different approach would be to allow the programmer to express QoS design intent in a very lightweight manner, not by invoking manually QoS-related APIs, but simply by opting to use OpenMP func-

tionality. In this model, the OpenMP runtime is in charge of automatically invoking low-level QoS calls without further programmer involvement.

In this work we focus on the latter approach, and leave the former for future work. The support for QoS at the programming model level has been integrated in a MPSoC-specific implementation of the OpenMP [183] compiler and runtime library based on the GCC 4.3 toolchain.

The idea here is to provide a sort of "privileged program area" for the execution of OpenMP threads, which are guaranteed not to be impacted by the effect of conflicting transactions from non-OpenMP programs. To do so, we embed in the parallel region manager of our runtime environment appropriate calls to the `setPriority()` and `resetPriority()` middleware API function presented in Section 4.4 on page 96. This has the effect of establishing prioritized channels towards the memory devices accessed by threads.

Each thread works on datasets which may change over time (i.e. at each parallel region). We exploit application profiling to ensure that each thread's data, ant any point in time, is accessible at the highest possible priority by appropriate QoS configuration of the NoC. More specifically, based on profile data priorities are automatically re-set at the beginning of each parallel region.

6.8.1 Balancing and Boosting Threads

In this section we describe the experimental setup that we considered to implement and evaluate the proposed framework. Our target platform template has been described in Figure 6.10(b) on page 159. We implemented an instance of this template within MPARM cycle accurate simulator [54]. Architectural details are reported in Table 6.4 on the following page.

To test the effectiveness of the approach, we employ a set of variants of the *Loop with dependencies* benchmark from the OmpSCR (OpenMP Source Code Repository suite [251]). In this program a number of parallelization schemes are considered to resolve loop carried dependencies. Due the possibility of finely tuning the workload through several parameters, as well as the alternation of communication-intensive and communication-intensive loops, this benchmark allows us to investigate a number of interesting case studies.

Typically, OpenMP programs achieve balanced parallel execution time between worker threads by allocating similar amounts of work to each of them. The most common example in this sense is loop parallelization with static scheduling, where the iteration space is evenly divided among

QoS-based NoC parameters	- 3x8 Quasi-Mesh - Fixed Priorities or Round Robin BE arbitration policy (if no priorization is performed) - Wormhole packet switching (FLIT_WITDH = 45 bits) - XY routing
Processing Elements	- 8 ARM RISC processor (4 KB data cache, 8 KB instruction cache) - QoS-NI initiator - Private local memories (placed at 0x00-0x07)
Shared memories	- Two banks of 8 shared memories (placed at 0x10-0x1F) - QoS-NI targets
Other	- Semaphore (placed at 0x20)

Table 6.4: Parameters of our OpenMP NoC-based MPSoC

threads. Communication is also evenly distributed: threads reference distinct equally-sized subsets of a shared data structure (i.e. an array).

From the architecture-agnostic application (or compiler) point of view, this is all that can be done to achieve workload balancing. Unfortunately, when mapping the application onto the hardware resources, many issues arise that can break balancing. High contention for the memory device where shared data is allocated may cause one (or more) thread(s) to be delayed in accessing its dataset. Due to the OpenMP semantics, where a barrier is implied at the end of each parallel region, this delay leads to overall program execution lengthening. Our aim has been to explore the effectiveness of high-priority packets in solving this issue.

To model the described problem, we allocate four of the eight available processors to the OpenMP program. The remaining four processors host independent threads that generate interfering/background traffic generated by other application. We consider three cases:

- All OpenMP threads within a communication-intensive loop are delayed in accessing shared data on a unique memory device.

- All OpenMP threads within a computation-intensive loop are delayed in accessing shared data on a unique memory device.

- Every OpenMP thread accesses a separate memory device: a single thread is delayed.

We describe each experiment in detail in the following sections.

6.8.2 Communication Dominated Loop

One of the techniques considered in our benchmark for loop parallelization in presence of dependencies is that of replicating the target array prior to overwriting its location with newly-computed data. The copy operation is fully parallel, with threads simply reading and writing to memory. In this experiment, OpenMP threads are hosted on processors four to seven. We allocate the target program arrays on a single memory device, namely shared memory 8. Interfering traffic to the same memory device is generated by noisy threads, running on processors zero to three.

Results for this experiments are reported in the plot on the left in Figure 6.17.

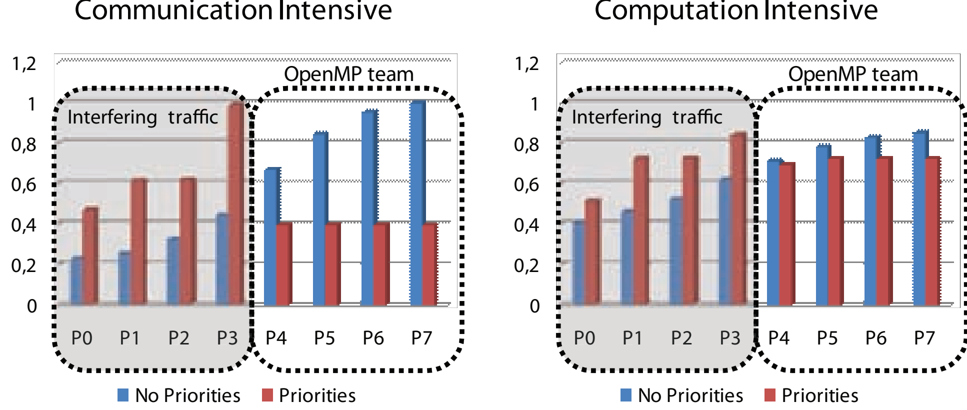


Figure 6.17: Effect of priorities on an OpenMP program unbalanced by memory contention

The x-axis represents processors zero to seven, whereas on the y-axis normalized execution time for each of them is reported. The blue bars sketch program behavior in absence of prioritized transactions, the red bars represent high priority transactions-enhanced OpenMP threads. It is possible to notice that in this mapping OpenMP threads experience very unbalanced execution, due to the effect in the interconnection medium of both memory device contention and arbitration policy. It is also possible to see that enclosing the OpenMP program within high priority threads allows achieving perfect balancing of workload.

Furthermore, program execution – whose performance degradation was originally dictated by the huge delay on processor 7 – is sped up by 60,25%. We also want to highlight a kind of ”system-level” effect of priorities. Interfering threads have assigned an amount of work similar to that of OpenMP

threads to show how the time recovered on processors four to seven is spread among non-prioritized code on processors zero to three.

6.8.3 Computation Dominated Loop

Once a copy of the original array content is stored in a replica, the second loop of the benchmark computes the new array element values by applying a computation-intensive kernel.

The results for this experiment are reported in the plot on the right in Figure 6.17 on the preceding page. Here the setup is largely similar to that described in the previous section. For this reason the behaviour of this loop closely resembles that of the previous one. Unsurprisingly, due to the reduced computation to communication ratio, here the effect of the high priority transactions is less pronounced. Nonetheless, a speedup of 14,87% is achieved on loop execution.

6.8.4 Balance Single Thread Interference

As a final experiment, we want to explore the role of priorities in a situation in which every OpenMP thread is accessing a separate memory device (thus no contention generates from the program itself). Here we suppose that a single thread is delayed in communicating with a memory by concurrent operations performed by other devices in the system (e.g. an accelerator performing DMA transfers or similar).

We model this situation in the following way. The OpenMP threads are mapped onto processors zero to three. Each of them executes the whole loop with dependencies benchmark, but we use a custom feature of our compiler that allows us to partition shared arrays in as many tiles as cores, and place the individual tiles onto nearby memories. Thus processor zero only accesses data on memory zero, processor one only accesses data on memory one, and so on. All of the interfering threads generate transactions to memory three, namely the memory accessed by OpenMP thread three. Looking at the blue bars in Figure 6.18 on the next page, it is possible to notice that this placement seriously degrades the performance of the OpenMP program, since thread three is delayed by the interfering transactions.

Even in this case it is possible to see that priorities completely solve the issue. The red bars show that the prioritized OpenMP program achieves perfectly balanced execution, distributing its original delay over non-prioritized threads. The overall program execution time achieves a 66,04% speedup.

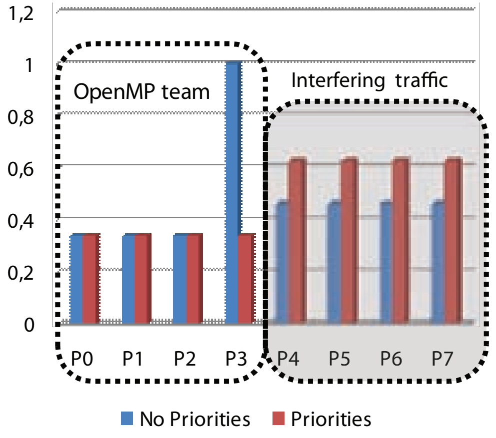


Figure 6.18: Effect of priorities on an OpenMP program unbalanced by a delayed thread

6.9 Conclusion and Open Issues

Through this approach we will be able to program efficiently homogeneous or heterogeneous NoC-based MPSoCs systems by using a high-level parallel programming models, and speedup applications and threads.

We present a complete software stack in order to exploit effectively all available computational resources included in our on-chip platform by adapting high-level MPI to our on-chip microtask communication message passing programming model.

Thus, we used this software stack to map high-level message passing parallel applications on a generic distributed-memory 2D-Mesh NoC-based MPSoC architecture, which it includes several Nios II soft-cores. Because the standard MPI is not suitable for on-chip systems, due to its large memory footprint and its high execution latencies, in this work we face an extremely accurate porting process without relying on an OS, on the design of a lightweight message passing interface library. This leads to a compact implementation version of the standard MPI by minimizing internal data structures, giving only support for basic MPI datatypes and operations, and defining only the useful functions to an efficient microtask communication in on-chip environments.
The obtained results in this chapter demonstrate three main issues:

- Mixing well-known modern on-chip architectures (i.e. NoC-based MPSoC), together with a standard message passing programming

model produces a reusable, robust an efficient microtask communication for NoC-based MPSoC.

- Our tiny ocMPI software library for microtask communication is a custom instance of MPI, and it is a viable solution to program NoC-based MPSoC architectures through a high-level message passing programming model.
- Due to its intrinsic support for heterogeneous systems, our ocMPI is a suitable way to program the future homogeneous or heterogeneous NoC-based MPSoC.

Finally, our ocMPI eases the design of coarse-grain or fine-grain parallel on-chip applications using the explicit parallelism and synchronization inherent in all MPI source codes. Moreover, MPI software designers do not need reeducation to write and run parallel applications in these NoC-based MPSoC architectures. This maximizes the productivity of software developers and the reuse of parallel applications.

As a future work, the ocMPI software library will evolve by adding more standard MPI functions useful for on-chip high-performance computing applications. The candidates are the non-blocking point-to-point functions and other collective communication routines. The ocMPI evolution also involves the optimization of the data distribution in most collective communication routines (now `ocMPI_Bcast()` sends *N-1* packets and `ocMPI_Barrier()` uses a inefficient master-slave scheme) using for instance a binary tree or butterfly distribution algorithms, which have faster completion times, typically O($log_2 N$).

Additionally, we will continue our research efforts to support custom hardware IP core modules which participate as stand alone peers in the ocMPI NoC-based system, instead of a coprocessor inside each tile. For this approach, a hardware ocMPI protocol must be embedded on the IP core.

On the other side, in this chapter, we introduced a complete framework to support QoS-related features at the programming model level. More specifically, we employ a layered design to expose the hardware QoS features of a NoC-based MPSoC architecture to the runtime environment of our OpenMP programming framework. This allows to build an infrastructure to assign varying levels of priority to the communication issued by OpenMP tasks.

Based on both a low-cost hardware design and a streamlined software implementation of the middleware API, we achieved an extremely efficient support for QoS. The results confirm a very small hardware overhead to support packet-level QoS. Experiments on a set of representative OpenMP

kernels show that, under different traffic patterns and thread allocation schemes, the use of prioritized transactions balance and boosts the overall execution time by up to 66%.

Future work on this field will focus on further extending the OpenMP support to QoS-related features. We are exploring the possibility of providing means for the programmer to explicitly specify prioritized execution within the program (e.g. with the use of specific compiler directives). Dynamic priority adjustments entail negligible overhead even at the single read/write level. Therefore the design overhead vs. performance gain of very fine grain QoS tuning is open for future study.

For all of these reasons, the outcome of this work will be cluster-on-chip environment which can be programmed using OpenMP or MPI for high-performance embedded computing (HPEC).

The discussion, and larger questions driving in this chapter is related to the software implication on a NoC-based MPSoCs. While there is a considerable large body of research in parallel programming models or software dependent stacks on cache coherent processors, in this dissertation, we believe that the advantages of message passing software in naturally distributed NoC-based MPSoCs are too great to be ignored.

First, managing messages and explicitly distributing data structures adds considerable complexity to the software development process. This complexity, however, is balanced by the greater ease of avoiding race conditions when all sharing of data is through explicit messages rather than through a shared address space.

Second, it is quite difficult to write software with predictable performance when the state of the cache is not predictable at all. A NoC-based MPSoC without cache coherence between cores avoids this problem.

Third, there are software engineering advantages to message passing architectures to model the software application. Software modules often are built upon the idea, that computing occurs on a "black box" entity, and the exchange of information only happen through the interfaces of it. A message passing architecture and its parallel programming model, like MPI, naturally supports this, and shared data only occurs when explicitly messages are transmitted and received.

Despite this, definitely, as a future work, the idea is to perform a fair comparison between both parallel programming models using a set of generic NoC-based MPSoC architecture, and benchmarks, in order to shed light on this open issue at the moment of program a multi-core system.

CHAPTER 7

Conclusions and Future Work

This short chapter reviews the initial motivations, shows the general conclusions related to this Ph.D dissertation, and afterwards, it lists open issues and potential future research directions on the area following this research.

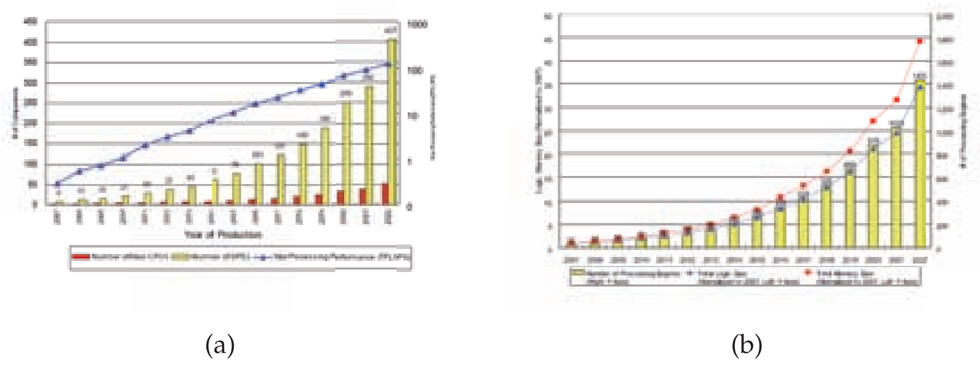
7.1 Overview and General Conclusions

Emerging consumer applications demand high performance and, specially when we are in the embedded domain, low-power and low-cost systems. The research, as well as the industry community, moved to design homogeneous or heterogeneous multi-core/many-core/MPSoC to fulfill the demands for next-generation SoCs.

Thus, we are witnessing the continuous growing on the number of processing elements and internal memory on the current situation and future multi-core systems (i.e. many-core or MPSoCs). In Figure 7.1 on the following page, we show the actual, and the prediction made by *The International Technology Roadmap for Semiconductors* in terms of number of components, internal memory, and the required performance on the upcoming years.

Thus, we are facing a big change, on hardware and software design, and attending on the revolution towards parallel processing, since future platforms will integrate multiple cores. We are still following Moore's law (doubling the number of transistors every 2 years), and shifting from the dead micro-architecture's Moore's law (double the performance per core every 2 years) towards a multi-core's Moore's law. This new Moore's law says that the number of cores per chip doubles every 2 years, and by extension it also claims to double the parallelism per workload every 2 years in conjunction with their architectural support, resulting on doubling the final performance per chip every 2 years.

In fact, as shown in Figure 7.1(a), the prediction shows that in 5 years time frame, i.e. in 2015, we will have to deal with hundreds of processors between main CPUs, and processing elements on SoC architectures which should deliver around 10 TFLOPS. In addition, as shown in Figure 7.1(b), the amount of memory will also increase in concordance with them, which opens many opportunities to design efficient memory hierarchies.



(a) (b)

Figure 7.1: SoC consumer design complexity trends (extracted from [1])

Considering that multi-core platforms are here, this dissertation provides a set of contributions to the scientific (exploring most degrees of freedom of NoC design space, see Figure 7.2(a) on the next page), and hopefully to the industry community, to face multi-core and many-core key design challenges:

- Design suitable NoC interconnections through EDA tools in order to generate energy efficient embedded multi-core systems.
- Simulation and validation environments, as well as real FPGA prototyping of NoC-based MPSoCs systems using different soft-core processors architectures (i.g. Nios II and ARM Cortex-M1).
- Design NoC interfaces to intercommunicate MPSoCs with heterogeneous standard protocols, such as OCP-IP and AMBA 2.0 AHB.
- Design and exploration of an AMBA AHB standalone FPU accelerator to speedup floating point operations, which can be also shareable among several processors, depending on system requirements.
- A reasonable low-cost hardware-software runtime guaranteed services QoS support for NoC-based systems.

- An efficient lightweight message passing parallel programming model and its specific hardware support to enable low latency inter-process communication, to enable parallel computing, and speedup parallel applications on a chip.

- Exploration on supporting QoS on top of OpenMP parallel programming model to boost critical threads, balance traffic, and meet application requirements, on a cluster on chip.

This Ph.D dissertation explores the three main key challenges to design full featured high performance embedded multi-core systems, at both, hardware and software level: *(i)* the interconnection backbone based on NoC paradigm through our NoCMaker EDA tool [50, 51], *(ii)* the hardware-software interfaces and to provision QoS through middleware routines [252], to enable runtime reconfiguration of hardware features, and *(iii)* the adaptation and/or customization of well-know parallel programming models, in our case MPI [129, 253, 254] to enable embedded parallel computing.

As is shown in Figure 7.2(b), this dissertation also states that using and combining well-know concepts, such as general theory on communication networks, with emerging many-core MPSoC and NoC-based architectures, in conjunction with parallel programming models, it generates new knowledge to face the trends on the design and prototyping of cluster on-chip multi-core architectures.

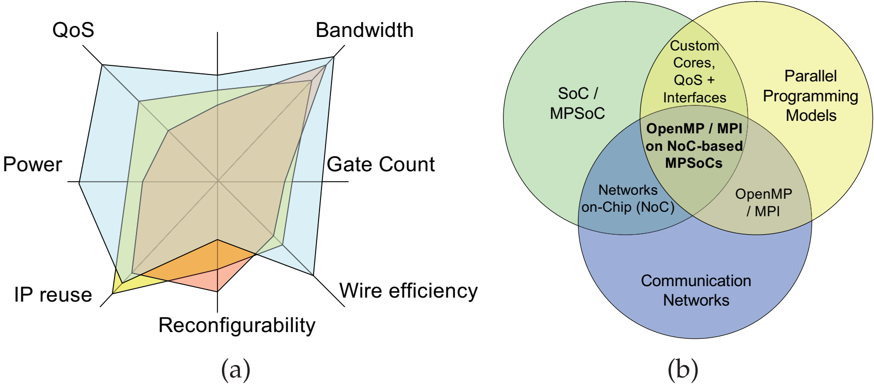


Figure 7.2: Thesis overview and NoC design space overview

Using the initial HW-SW methodology (see Figure 1.4 on page 14), and thanks to the presented HW-SW components developed and presented along this Ph.D thesis, multiple scalable multi-core and MPSoC architec-

tures have been designed by exploiting hardware features using software facilities.

More in detail, in Chapter 2, this dissertation has demonstrated that, using xENoC – NoCMaker as EDA tool, we can reach a rapid and correct-by-construction prototyping of different interconnection fabrics for next-generation NoC-based MPSoCs, as well as a robust simulation and validation environment to obtain early estimations in terms of area costs, power consumption and performance. We demonstrated that our variant of circuit switching uses around 15% less area than a conventional packet switching with minimal buffer capabilities. We also shown trade-offs between common regular topologies, giving the ratio between the communication and the computational resources, in several embedded parallel Nios II NoC-based MPSoC architectures. The tool flow will enhance the design productivity and bridge the gap between computation and communication architecture.

In Chapter 3, we shown the design of a full featured AMBA 2.0 AHB NI, which is OCP-IP compatible. The results shown that, our NI design is slim and efficient. The area overhead is around 25% (which does not impact a lot on the final size of the NoC backbone) compared to the equivalent OCP-IP NI. This is mainly because of the fact that, it is required to adapt AMBA 2.0 AHB to OCP-IP signals, and the pipelined nature of AMBA 2.0 AHB. On the other side, the circuit performance only drops $\sim$1,5%, that can definitely be considered negligible. The main benefit of this work is that enables the reusability, enabling the transparent plug&play of pre-designed and pre-tested IP cores compliant with these two on-chip communication protocols. Thus, on the final NoC-based system, all IP blocks will be fully compatible at transactional level from the user point of view. This is a key challenge to boost design productivity, and fast time-to-market, at the moment of design multi-protocol heterogeneous multi-core systems.

In Chapter 4, we provisioned runtime QoS (i.e. a traffic class classification using a unified buffer priority scheme, and guaranteed services using the emulation of circuit switching on wormhole packet switching NoC backbone) on AMBA AHB and OCP Network Interfaces at NoC level through middleware software routines. On average, the area overhead to support QoS from middleware software routines at NI level is about 20% on area, and without any influence on circuit performance (i.e. f_{max}). Despite this fact, at switch level, the inclusion of QoS features shows an important impact, specially when the number of priority levels increase, in terms of area and performance. Thus, our tests assess an area overhead of 25% and 40-50% (depending on the target device), when we support 2 and 4 levels of

priority, respectively. Furthermore, this impact raises up dramatically until 100% (i.e. doubling the area of a switch without QoS) if 8 levels of priority are supported on the switch. On the other side, on average, the circuit performance drops also significantly on the tested FPGA devices, i.e. 15-20%, 24-29% and 35% with 2, 4, and 8 priority levels, respectively. Despite this fact, the benefits are big since we can handle and balance traffic, or guaranteed throughput at runtime, when congestion and hotspots appear on the NoC backbone.

In Chapter 5, we explored the way to accelerate floating point instructions designing a standalone memory mapped (either as a NoC tile or crossbar element) FPU accelerator, exploring the different HW-SW notification protocols to enable HW assisted FP instructions. Moreover, we also investigated their shareability among different ARM Cortex-M1 processors on a full featured cluster on chip. Final results demonstrate execution speedups ranging from ∼3x to ∼45x compared with an equivalent ARM emulated floating point software library. At only 50 MHz, the cluster on chip platform delivers 1.1-3.5 MFLOPs, depending on the CPU-FPU notification protocol used.

In Chapter 6, an efficient lightweight MPI-based parallel programming model SW stack have been designed and targeted to inter-process communication on a chip. The software stack with the most common MPI communication functions only requires ∼13 KB, which makes it suitable for embedded multi-core systems. Inter-process communication latencies have been profiled under different workloads, as well as for barrier synchronization, quantified around tens of μs to distribute small data sets running the system at 100 MHz. Traditional highly parallel applications have been parallelized resulting on speedups close to the number of processors. This demonstrates its feasibility to be the parallel programming model of future distributed NoC-based MPSoCs or many-core architectures. The source codes of parallel applications have been compiled and executed without any relevant changes from the source MPI codes used in any traditional large-scale distributed system, which enables software code reuse, and the unnecessary re-education of software developers.

On the other side, along Chapter 6, we improved OpenMP kernels applying QoS on top of the shared memory parallel programming model targeted to embedded systems. Thus, we are able to balance, boost threads, and definitely to fulfill application requirements, when interfering workloads use an specific resources (in our case the shared memories) which are also used by the runtime OpenMP software layer. The outcome are execution speedups up to 66% in traditional OpenMP loop parallelization.

Even if further investigation should be done, the author believe that this Ph.D dissertation, with all the engineering contributions, is a first step towards the required software driven flexibility enhancing its programmability through high-level parallel programming models to achieve high-performance embedded computing.

7.2 Open Issues and Future Work

As described before, at the end of each chapter, there are few specific future work according to each specific topic presented in this Ph.D dissertation.

In Chapter 2, the main open issue is to extend our EDA tool, i.e. NoC-Maker, in order to generate application-specific NoC-based MPSoC architectures.

In Chapter 3, the most important enhancement as future work, it will be to improve NIs to handle as much as possible inter-protocol interoperability and compatibility (i.e AHB, AXI, OCP, etc). This will increase even more IP reuse, as well as transparent plug&play of commodity IPs of different protocols to design next generation NoC-based SoCs.

In Chapter 4, as future research, the idea is to provision runtime monitoring, DVFS and dynamic power and thermal management services applying the same philosophy presented on the chapter, which is focus to provide exclusively runtime QoS using hardware support and software middleware routines.

In Chapter 5, as future work, the main target is to expand the functionality of the standalone single precision FPU accelerator to support double precision floating point operations, and study more in deep the configuration of the register file to enable pipeline transactions using the intercommunication fabric (i.e. hierarchical bus or NoC backbone).

In Chapter 6, as future research, the idea is to explore the hybrid parallelization using OpenMP and MPI, and potential extensions and their interaction to enable a better synergy with the next generation of embedded parallel heterogeneous architectures.

The industry had manufactured different experimental research prototypes of many-core architecture using regular 2D Mesh NoC backbones (e.g. Ambric [255,256], Tilera [257], and 80-core Polaris [200,258]). Very recently, Intel launched within *Intel's Tera-scale Computing Research Program* [259], a fully compliant x68 Intel Architecture 48-core Single-Chip Cloud Computer (SCC) [260], and Larrabee GPGPU [261] for visual computing, which are an evolution of its predecessor Polaris.

Even if, the industry manufacture experimental many-core chips, there

is still a big hole in research, to enable the easy programmability and reconfigurability of these many-core chips through high-level parallel programming models. As proposed in this Ph.D dissertation, some of them include hardware support to enable message passing communication and synchronization models and equivalent stream-based software stacks.

However, this open issue, will be even more challenging where the multi-core execution platform tends to be asymmetric or more heterogeneous. As a consequence, more research and development must be done in this direction, in order to define potentially hybrid or adaptive parallel programming models and runtime layers, which can exploit on different scenarios, all features present on the hardware execution platform.

In Figure 7.3, we highlight the prediction done by *The International Technology Roadmap for Semiconductors*, which clearly list future directions depicting the increasing importance of software costs in the future for SoC design. Thus, in next years we will shift towards heterogeneous parallel processing, and more far to system design automation, going through by the generation of HW-SW intelligent testbenchs and runtime debuggers at higher levels of abstraction, since verification is still consuming most of the time at the moment to design complex HW-SW multi-core systems.

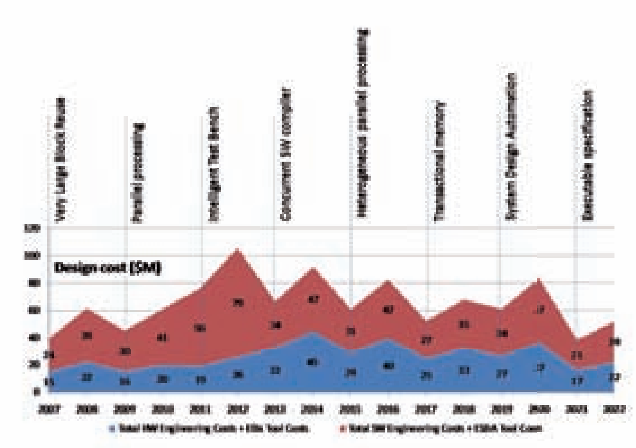


Figure 7.3: HW-SW design costs on future SoC platforms (from [1])

Thus, there are several exciting and challenging long term open issues and research lines that can be performed specially on the parallel programming software area. In this section, we summarize the current works in progress and future research directions:

- Runtime reconfiguration of NoC services (thermal and power man-

agement, performance monitoring and debugging, service discovery, reliability and fault tolerance, ...) using hardware support and middleware software libraries.

- Hybrid programming of future 2D/3D NoC-based MPSoC or many-core chips.
- Intelligent functional verification, and runtime system level debugging through hardware and software components.
- Adapting existing debugging, and performance analysis HPC tools in conjunction with the presented parallel programming models for the embedded domain.

Thus, future parallelization on heterogeneous multi-core execution platforms must be capable of dynamically adapting once the application is compiled on the underlying hardware (2D, 3D chips), by changing the number of cores, number of function units, modifying the NoC design space parameters, changing between power-modes, as well as track potential HW or SW bugs, etc. In addition, to the static parallelization framework demonstrated in this thesis, an OS or a runtime system layer will be required to enable runtime application reconfiguration over time, re-evaluating the initial static parallelism strategy, and execution decisions, whenever it is required.

APPENDIX A

Author's Relevant Publications

The appendix lists the works in which the author has been involved and which are relevant to the dissertation's subject.

- **Jaume Joven et al.**, ”Shareable Compiler-unaware FPU Accelerator on an Extensible Symmetric Cortex-M1 Cluster on-a-Chip”, *To be submitted to DSD 2010*.

- **Jaume Joven et al.**, ”Application-to-Packets QoS Support - A Vertical Integrated Approach for NoC-based MPSoC”, *To be resubmitted on CODESS+ISSS 2010*.

- **Jaume Joven**, ”A Lightweight MPI-based Programming Model and its HW Support for NoC-based MPSoCs”, *PhD Forum DATE, IEEE/ACM Design, Automation and Test in Europe (DATE'09)*, April 2009, Nice, France.

- **Jaume Joven**, David Castells-Rufas, Sergi Risueo, Eduard Fernandez, Jordi Carrabina, ”NoCMaker & j2eMPI - A Complete HW-SW Rapid Prototyping EDA Tool for Design Space Exploration of NoC-based MPSoCs”, *IEEE/ACM Design, Automation and Test in Europe (DATE'09)*, April 2009, Nice, France.

- David Castells-Rufas, **Jaume Joven**, Sergi Risueo, Eduard Fernandez, Jordi Carrabina, ”NocMaker : A Cross-Platform Open-Source Design Space Exploration Tool for Networks on Chip”, *3th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC'09)*, January 2009, Paphos, Cyprus.

- **Jaume Joven**, David Castells-Rufas, Jordi Carrabina, "An Efficient MPI Microtask Communication Stack for NoC-based MPSoC Architectures", *Fourth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'08)*, July 2008, L'Aquila, Italy.

- **Jaume Joven**, Oriol Font-Bach, David Castells-Rufas, Ricardo Martinez, Lluis Teres, Jordi Carrabina, "xENoC - An eXperimental Network-On-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC architectures", *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'08)*, February 2008, Toulouse, France (**best paper award)**.

- **Jaume Joven**, David Castells-Rufas, Jordi Carrabina. "HW-SW Framework for Parallel Distributed Computing on NoC-based MPSoC architectures", *Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07)*, July 2007, L'Aquila, Italy.

- **Jaume Joven**, Jorge Luis Zapata, David Castells-Rufas, Jordi Carrabina. "Embedded Parallel Software Framework for on-chip Distributed Parallel Computing", *XIII Taller Iberchip (IWS'07)*, March 2007, Lima, Peru.

- David Castells-Rufas, **Jaume Joven**, Jordi Carrabina. "A Validation and Performance Evaluation Tool for ProtoNoC". *IEEE International Symposium on System-on-Chip (ISSoC'06)*, November 2006, Tampere, Finland.

- **Jaume Joven**, David Castells-Rufas, Jordi Carrabina. "HW-SW Framework for Distributed Parallel Computing on Programmable Chips". *XXI Conference on Design of Circuits and Integrated Systems (DCIS'06)*, ISBN: 978-84-690-4144-4, November 2006, Barcelona, Spain.

- **Jaume Joven**, David Castells-Rufas, Jordi Carrabina. "Metodologa de Codiseo Hardware-Software para la Computacin Paralela Distribuida dentro de un chip". *VI Jornadas de Computacin Reconfigurable y Aplicaciones (JCRA'04)*, September 2006, Caceres, Spain.

APPENDIX B

Curriculum Vitae

Jaume Joven Murillo was born on 12th April 1981, in Terrassa (Barcelona), Spain. He completed his high school and junior college education in Escola Pia de Terrassa, in 1999.

After he started to pursue his graduate education, obtaining his Master of Science in Computer Science in 2004 for *"Components i Sistemes SoC per a bus AMBA i Especificació XML"* at Universitat Autonòma de Barcelona (UAB). At the same he worked as HW-SW engineer and assistant researcher in CEPHIS Lab in cooperation with EPSON Electronics Barcelona Design Center (EPSON-BDC).

In September 2005, he began his post-graduate, in the Ph.D. program in Computer Science (specialization in Electronic and Microelectronics Circuits and Systems) at Universitat Autònoma de Barcelona (UAB). Afterwards, he received a Ph.D. grant awarded by Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR) in conjunction to D+T Microelectrónica A.I.E, company associated to Centre Nacional de Microelectrónica at Institut de Microelectrónica de Barcelona (CNM-IMB).

In April 2007, he received his Master of Advanced Studies (MAS) for his work about *"HW-SW Framework for Parallel Distributed Computing on NoC-based MPSoC architectures"* at Universitat Autònoma de Barcelona (UAB).

During the Ph.D. he had the privilege to work for three months (Sep-Dec 2008) on the R&D Department of a world-class company as is ARM Ltd. in Cambridge, being the winner of a contests to get the ARM Ph.D. intern

position. Moreover, during his research, he spent two periods of 6 months (Feb-Jul 2008-2009) as a visiting Ph.D. student at Integrated Systems Laboratory (LSI) at Ecole Polytechnique Fédérale de Lausanne (EPFL) thanks to a grant BE-2 awarded by AGAUR, and HIPEAC Ph.D. collaboration grant, respectively.

Along his Ph.D., he had also teaching experience, working sporadically at UAB as an Assistant Professor (2007-2008), giving subjects in Computer Science, Telecommunications and Electronics Degrees.

Finally, he received the Ph.D. for this dissertation titled *"HW-SW Components for Parallel Embedded Computing on NoC-based MPSoCs"*, in January 2010 at Universitat Autònoma de Barcelona (UAB).

He held various positions along the last 5 years, mostly as embedded software engineer, hardware engineer and system designer. Through his roles, he gained extensive experience in the field of computer engineering, microelectronics and electronic design, as well as in telecommunications and communication systems.

Glossary

3G Third-generation - Is a third generation of mobile phone standards and technology, after 2G. It is based on the International Telecommunication Union (ITU) family of standards under the International Mobile Telecommunications programme, IMT-2000. 3G technologies enable network operators to offer users a wider range of more advanced services while achieving greater network capacity through improved spectral efficiency. Services include wide-area wireless voice telephony and broadband wireless data, all in a mobile environment.

ALU **A**rithmetic and **L**ogic **U**nit - Component of the central processing unit, CPU that performs calculations and logical operations.

API **A**pplication **P**rogramming **I**nterface - Is the interface that a computer system, library or application provides in order to allow requests for services to be made of it by other computer programs, and-or to allow data to be exchanged between them.

ASIC **A**pplication-**S**pecific **I**ntegrated **C**ircuit - Is is an integrated circuit (IC) customized for a particular applications rather than intended for general-purpose use.

ASIP **A**pplication **S**pecific **I**nstruction-Set **P**rocessor - Is a methodology used in SoC design. It represents a compromise between ASIC and general purpose CPU. In ASIP, the instruction set provided by the core can be configured to fit the specific application. This configurability of the core provides a trade-off between flexibility and performance. Usually, the core is divided into two parts: static logic which defines a minimum ISA and configurable logic which can be used to design new instructions.

CMP **C**hip-**L**evel **M**ultiprocessing - In computing, chip-level multiprocessing is symmetric multiprocessing (SMP) implemented on a single VLSI integrated circuit. Multiple processor cores (multicore) typically share a common second- or third-level cache and interconnect.

CMT **C**hip **m**ulti**t**hreading - Is the capability of a processor to process multiple software threads and supports simultaneous hardware threads of execution. CMT is achieved by having multiple cores on a single chip (to share chip resources such as, the memory controller and the L2 cache) or multiple threads on a single core.

DMA **D**irect **M**emory **A**ccess - Is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel

DSM **D**eep **S**ub-**M**icrometre Technology.

DSP **D**igital **S**ignal **P**rocessor - Is a specialized microprocessor designed specifically for digital signal processing (sound, video streaming...) generally in real-time.

DVFS **D**ynamic **V**oltage **F**requency **S**caling - Is a technique in computer architecture whereby the frequency of a microprocessor can be automatically adjusted on-the-fly, either to conserve power or to reduce the amount of heat generated by the chip.

EDA **E**lectronic **D**esign **A**utomation - is the category of tools for designing and producing electronic systems ranging from printed circuit boards (PCBs) to integrated circuits (ICs). This is sometimes referred to as ECAD (electronic computer-aided design) or just CAD.

ESL **E**lectronic **S**ystem **L**evel - An emerging electronic design methodology for SoCs which focuses on the higher abstraction level concerns first and foremost. ESL is evolving into a set of complementary methodologies that enable embedded system design, verification, and debugging through to the hardware and software implementation of custom SoC, system-on-FPGA, system-on-board, and entire multi-board systems. ESL can be accomplished through the use of SystemC as an abstract modeling language.

FFT **F**ast **F**ourier **T**ransform - Is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFTs are of great importance to a wide variety of applications, from digital signal processing to solving partial differential equations to algorithms for quickly multiplying large integers.

FPGA **F**ield **P**rogrammable **G**ate **A**rray - A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders

or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories.

FPU **F**loating **P**oint **U**nit - Is a part of a computer system specially designed to carry out operations on floating point numbers. Typical operations are floating point arithmetic (such as addition and multiplication), but some systems may be capable of performing exponential or trigonometric calculations as well (such as square roots or cosines).

FSM **F**inite **S**tate **M**achine - Is is a model of behavior composed of a finite number of states, transitions between those states, and actions.

GALS **G**lobally **A**synchronous **L**ocally **S**ynchronous **C**ircuits - Is a system which do not have a global clock, but wherein each submodule is independently clocked.

GPS **G**lobal **P**ositioning **S**ystem - Is the only fully functional Global Navigation Satellite System (GNSS). Utilizing a constellation of at least 24 medium Earth orbit satellites that transmit precise microwave signals, the system enables a GPS receiver to determine its location, speed, direction, and time.

GUI **G**raphical **U**ser **I**nterface - Is a type of user interface which allows people to interact with a computer and computer-controlled devices which employ graphical icons, visual indicators or special graphical elements, along with text, labels or text navigation to represent the information and actions available to a user.

HDL **H**ardware **D**escription **L**anguage - In electronics, a hardware description language or HDL is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design, and tests to verify its operation by means of simulation.

HSDPA **H**igh-**S**peed **D**ownlink **P**acket **A**ccess - Is a 3G (third generation) mobile telephony communications protocol in the High-Speed Packet Access (HSPA) family, which allows networks based on Universal Mobile Telecommunications System (UMTS) to have higher data transfer speeds and capacity.

IC **I**ntegrated **C**ircuit - A monolithic integrated circuit is a miniaturized electronic circuit (consisting mainly of semiconductor devices, as well as passive components) which has been manufactured in the surface of a thin substrate of semiconductor material.

IP **I**ntellectual **P**roperty core - Is a reusable hardware component, represented either as synthesizable HDL, or as hardware netlist, or as layout, part of a system-on-chip.

ISA **I**nstruction **S**et **A**rchitecture - Is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external IO. An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular CPU design.

LUT **L**ook-**u**p **T**able - Is a data structure, usually an array or associative array, used to replace a runtime computation with a simpler lookup operation. The speed gain can be significant, since retrieving a value from memory is often faster than undergoing an expensive computation.

NoC **N**etwork-**o**n-**C**hip - Is a new approach to design the communication subsystem of SoCs. NoC-based systems can accommodate multiple asynchronous clocking that many of today's complex SoC designs use. NoC brings networking theories and systematic networking methods to on-chip communication and brings notable improvements over conventional bus systems. NoC greatly improve the scalability of SoCs, and shows higher power efficiency in complex SoCs compared to buses.

NUMA **N**on **U**niform **M**emory **A**ccess - Is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

OCB **O**n-**c**hip **B**us - Term to define hierarchical or shared integrated on-chip communication bus used in SoCs in order to interconnect IP cores (e.g. AMBA, Avalon, IBM CoreConnect, Wishbone, STBus, etc).

ocMPI **o**n-**c**hip **M**essage **P**assing **I**nterface - Lightweight version of MPI presented in this dissertation targeted to the emerging NoC-based MPSoCs.

QoS **Q**uality **o**f **S**ervice - In the fields of packet-switched networks and computer networking, the traffic engineering term Quality of Service, refers to resource reservation control mechanisms. Quality of Service can provide different priority to different users or data flows, or guarantee a certain level of performance to a data flow in accordance with requests from the application program or the internet service provider policy.

RTL **R**egister **T**ransfer **L**evel - Is a way of describing the operation of a digital circuit. In RTL design, a circuit's behavior is defined in terms of the flow of signals or transfer of data between registers, and the logical operations performed on those signals. RTL is used in HDLs like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can then be derived.

SDR **S**oftware-**D**efined **R**adio - Is a radio communication system which can tune to any frequency band and receive any modulation across a large frequency spectrum by means of a programmable hardware which is controlled by software. An SDR performs significant amounts of signal processing in a general purpose computer, or a reconfigurable piece of digital electronics. The goal of this design is to produce a radio that can receive and transmit a new form of radio protocol just by running new software.

SMP **S**ymmetric **M**ulti**p**rocessing - Is a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory.

SoC **S**ystem-**o**n-**C**hip - Is an idea of integrating all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions all on one chip. A typical application is in the area of embedded systems.

TDMA **T**ime **D**ivision **M**ultiple **A**ccess - Is a channel access method for shared medium (usually radio) networks. It allows several users to share the same frequency channel by dividing the signal into different timeslots.

TLM **T**ransaction-**L**evel **M**odeling - Is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. At the transaction level, the emphasis is more on the functionality of the data transfers - what data are transferred to and from what locations - and less on their actual implementation, that is, on the actual protocol used for data transfer.

UART **U**niversal **A**synchronous **R**eceiver/**T**ransmitter - Is a type of asynchronous receiver/transmitter, a piece of computer hardware that translates data between parallel and serial interfaces. Used for serial data telecommunication (i.e. serial ports), a UART converts bytes of data to and from asynchronous start-stop bit streams represented as binary electrical impulses.

UMA **U**niform **M**emory **A**ccess - Is a computer memory architecture used in parallel computers having multiple processors and probably multiple memory chips. All the processors in the UMA model share the physical memory uniformly. Cache memory may be private for each processor. In a UMA architecture, accessing time to a memory location is independent from which processor makes the request or which memory chip contains the target memory data.

UML **U**nified **M**odelling **L**anguage - In software engineering, is a non-proprietary specification language for object modelling. UML is a general-purpose modelling language that includes a standardized graphical notation used to cre-

ate an abstract model of a system, referred to as a UML model. UML is extendable as it offers a profile mechanism for customization.

UMTS **U**niversal **M**obile **T**elecommunications **S**ystem - Is one of the third-generation (3G) cell phone technologies. Currently, the most common form uses W-CDMA as the underlying air interface, is standardized by the 3GPP, and is the European answer to the ITU IMT-2000 requirements for 3G cellular radio systems.

VLIW **V**ery **L**ong **I**nstruction **W**ord - It refers to a CPU architectural approach to taking advantage of instruction level parallelism (ILP) The performance can be improved by executing different sub-steps of sequential instructions simultaneously, (this is pipelining), or even executing multiple instructions entirely simultaneously as in superscalar architectures. Further improvement can be realized by executing instructions in an order different from the order they appear in the program; this is called out of order execution.

Bibliography

[1] "The International Technology Roadmap for Semiconductors," Tech. Rep., 2007, http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[2] L. Teres, Y. Torroja, S. Olcoz, and E. Villar, *VHDL, Lenguaje Estndar de Diseo Electrnico*. McGraw-Hill/Interamericana de España, 1998.

[3] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the 38th Design Automation Conference*, June 2001, pp. 684–689.

[4] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[5] J. pekka Soininen, A. Jantsch, M. Forsell, A. Pelkonen, J. Kreku, and S. Kumar, "Extending Platform-Based Design to Network on Chip Systems," in *Chip Systems, Proceedings International Conference on VLSI Design*, 2003, pp. 46–55.

[6] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.

[7] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701–1713, Oct. 2008.

[8] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[9] J. Hu, Y. Deng, and R. Marculescu, "System-Level Point-to-Point Communication Synthesis Using Floorplanning Information," in *Proc. 7th Asia and South Pacific and the 15th International Conference on VLSI Design Design Automation Conference Proceedings ASP-DAC 2002*, Jan. 7–11, 2002, pp. 573–579.

[10] M. Birnbaum and H. Sachs, "How VSIA answers the SOC dilemma," *IEEE Computer*, vol. 32, no. 6, pp. 42–50, June 1999.

[11] Virtual Socket Interface Alliance (VSIA), "On-chip Bus Attributes and Virtual Component Interface," http://www.vsi.org/.

[12] OCP International Partnership (OCP-IP), "Open Core Protocol Standard," 2003, http://www.ocpip.org/home.

[13] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *DATE '00: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2000, pp. 250–256.

[14] "ARM AMBA 2.0 AHB-APB Overview," ARM Ltd., 2005, http://www.arm.com/products/system-ip/interconnect/amba-design-kit.php.

[15] *AMBA Specification*, ARM Inc., May 1999, http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[16] "Aeroflex Gaisler," http://www.gaisler.com/.

[17] "IBM CoreConnect Bus Architecture," IBM, https://www-01.ibm.com/chips/techlib/techlib.nsf/pages/main.

[18] "Xilinx MicroBlaze Soft Processor core," Xilinx, http://www.xilinx.com/tools/microblaze.htm.

[19] "Avalon Interface Specifications," 2009, http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.

[20] "Altera Nios II Embedded Processor," Altera, http://www.altera.com/literature/lit-nio2.jsp.

[21] "Philips Nexperia - Highly Integrated Programmable System-on-Chip (MPSoC)," 2004, http://www.nxp.com.

[22] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems," *IEEE Design and Test of Computers*, vol. 18, no. 5, pp. 21–31, 2001.

[23] Texas Instruments (TI), "OMAP Platform," 2004, http://focus.ti.com/omap/docs/.

[24] "ST Nomadik Multimedia Processor," 2004, http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

[25] "The Cell project at IBM Research - Heterogeneous Chip Multiprocessing," 2004, http://www.research.ibm.com/cell/.

[26] W. Wolf, "The Future of Multiprocessor Systems-on-Chips," in *Proc. DAC*, June 2004, pp. 681–685.

[27] *Multi-layer AHB Technical Overview*, ARM Ltd., 2001, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dvi0045b/index.html.

[28] "STBus Interconnect," STMicroelectronics, http://www.st.com/.

[29] D. Wingard, "MicroNetwork-based Integration for SOCs," in *Proc. Design Automation Conference*, 2001, pp. 673–677.

[30] E. Salminen, T. Kangas, T. D. Hämäläinen, J. Riihimäki, V. Lahtinen, and K. Kuusilinna, "HIBI Communication Network for System-on-Chip," *J. VLSI Signal Process. Syst.*, vol. 43, no. 2-3, pp. 185–205, 2006.

[31] S. Pasricha, N. D. Dutt, and M. Ben-Romdhane, "BMSYN: Bus Matrix Communication Architecture Synthesis for MPSoC," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 8, pp. 1454–1464, Aug. 2007.

[32] "AMBA 3 AXI overview," ARM Ltd., 2005, http://www.arm.com/products/system-ip/interconnect/axi/index.php.

[33] F. Angiolini, P. Meloni, S. M. Carta, L. Raffo, and L. Benini, "A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MPSoCs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 3, pp. 421–434, Mar. 2007.

[34] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "MPEG-based Performance Comparison between Network-on-Chip and AMBA MPSoC," in *Proc. 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems DDECS 2008*, Apr. 16–18, 2008, pp. 1–6.

[35] C. Bartels, J. Huisken, K. Goossens, P. Groeneveld, and J. van Meerbergen, "Comparison of An Æthereal Network on Chip and A Traditional Interconnect for A Multi-Processor DVB-T System on Chip," in *Very Large Scale Integration, 2006 IFIP International Conference on*, Oct. 2006, pp. 80–85.

[36] Arteris, "A Comparison of Network-on-Chip and Busses," Arteris, Tech. Rep., 2005, http://www.arteris.com/noc_whitepaper.pdf.

[37] K. C. Janac, “When is the use of a NoC Most Effective and Why,” in *7th International Forum on Application-Specific Multi-Processor SoC*, 2007.

[38] E. Salminen, T. Kangas, V. Lahtinen, J. Riihimäki, K. Kuusilinna, and T. D. Hämäläinen, “Benchmarking Mesh and Hierarchical Bus Networks in System-on-Chip Context,” *J. Syst. Archit.*, vol. 53, no. 8, pp. 477–488, 2007.

[39] L. Benini and G. De Micheli, “Networks on Chips: A new SoC Paradigm,” *IEEE Computer*, vol. 35, no. 1, pp. 70 – 78, January 2002.

[40] A. Jantsch and H. Tenhunen, *Networks on chip*, K. A. Publishers, Ed. Hingham, MA, USA: Kluwer Academic Publishers, 2003.

[41] L. Benini and G. D. Micheli, *Networks on chips: Technology and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2006.

[42] G. Mas and P. Martin, “Network-on-Chip: The Intelligence is in The Wire,” in *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors ICCD 2004*, 2004, pp. 174–177.

[43] A. A. Jerraya and W. Wolf, “Hardware/Software Interface Codesign for Embedded Systems,” *Computer*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[44] J.-Y. Mignolet and R. Wuyts, “Embedded Multiprocessor Systems-on-Chip Programming,” *Software, IEEE*, vol. 26, no. 3, pp. 34–41, May-June 2009.

[45] D. Black and J. Donovan, *SystemC: From the Ground Up*. Kluwer Academic Publishers, 2004.

[46] “Open SystemC Initiative - Defining & Advancing SystemC Standards,” http://www.systemc.org/home/.

[47] “ModelSim® – Advanced Simulation and Debugging,” http://www.model.com.

[48] “Mentor Graphics® – Functional Verification,” http://www.mentor.com/products/fv/modelsim/.

[49] “GTKWave Electronic Waveform Viewer,” http://intranet.cs.man.ac.uk/apt/projects/tools/gtkwave/.

[50] J. Joven, O. Font-Bach, D. Castells-Rufas, R. Martinez, L. Teres, and J. Carrabina, "xENoC - An eXperimental Network-On-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC Architectures," in *Proc. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP 2008*, Feb. 13–15, 2008, pp. 141–148.

[51] D. Castells-Rufas, J. Joven, S. Risueo, E. Fernandez, and J. Carrabina, "NocMaker: A Cross-Platform Open-Source Design Space Exploration Tool for Networks on Chip," in *3th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip*, 2009.

[52] D. Bertozzi and L. Benini, "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip," vol. 4, no. 2, pp. 18–31, 2004.

[53] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. De Micheli, "×pipes Lite: A Synthesis Oriented Design Library for Networks on Chips," in *Proc. Design, Automation and Test in Europe*, 2005, pp. 1188–1193.

[54] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, September 2005.

[55] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An Asynchronous NoC Architecture Providing Low Latency Service and Its Multi-Level Design Framework," in *Proc. 11th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2005*, Mar. 14–16, 2005, pp. 54–63.

[56] D. T. Stephan Kubisch, Enrico Heinrich, "A Mesochronous Network-on-Chip for an FPGA," Tech. Rep., 2007, http://www.imd.uni-rostock.de/veroeff/memics2007_paper.pdf.

[57] I. Loi, F. Angiolini, and L. Benini, "Developing Mesochronous Synchronizers to Enable 3D NoCs," in *Proc. Design, Automation and Test in Europe DATE '08*, Mar. 10–14, 2008, pp. 1414–1419.

[58] U. Y. Ogras, J. Hu, and R. Marculescu, "Key research Problems in NoC Design: A Holistic Perspective," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2005, pp. 69–74.

[59] S. Murali and G. De Micheli, “An Application-Specific Design Methodology for STbus Crossbar Generation,” in *Proc. Design, Automation and Test in Europe*, 2005, pp. 1176–1181.

[60] L. Bononi and N. Concer, “Simulation and Analysis of Network on Chip Architectures: Ring, Spidergon and 2D Mesh,” in *Proc. Design, Automation and Test in Europe DATE '06*, vol. 2, Mar. 6–10, 2006, p. 6pp.

[61] S. Murali and G. De Micheli, “SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs,” in *Proc. 41st Design Automation Conference*, 2004, pp. 914–919.

[62] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, “Effect of Traffic Localization on Energy Dissipation in NoC-based Interconnect,” in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2005*, May 23–26, 2005, pp. 1774–1777.

[63] M. Saldaña, L. Shannon, and P. Chow, “The Routability of Multiprocessor Network Topologies in FPGAs,” in *SLIP '06: Proceedings of the 2006 international workshop on System-level interconnect prediction*, 2006, pp. 49–56.

[64] A. Hansson, K. Goossens, and A. Rădulescu, “A Unified Approach to Constrained Mapping and Routing on Network-on-Chip Architectures,” in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2005, pp. 75–80.

[65] J. Hu and R. Marculescu, “Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures,” in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, 2003.

[66] S. Bourduas and Z. Zilic, “A Hybrid Ring/Mesh Interconnect for Network-on-Chip Using Hierarchical Rings for Global Routing,” in *Proc. First International Symposium on Networks-on-Chip NOCS 2007*, May 7–9, 2007, pp. 195–204.

[67] T. Bjerregaard, “The MANGO Clockless Network-on-Chip: Concepts and Implementation,” Ph.D. dissertation, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005, supervised by Assoc. Prof. Jens Sparsø, IMM. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?4025

[68] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, “Designing Application-Specific Networks on Chips with Floorplan Information,” in *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 355–362.

[69] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, “×pipesCompiler: A Tool for Instantiating Application-Specific Networks on Chip,” in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, 2004, pp. 884–889.

[70] L. Benini, “Application specific NoC Design,” in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 491–495.

[71] A. Mello, L. C. Ost, F. G. Moraes, and N. L. Calazans., “Evaluation of Routing Algorithms on Mesh Based NoCs,” Technical Report TR2000-02, University of Texas at Austin, USA, Tech. Rep., 2004, http://www.inf.pucrs.br/tr/tr040.pdf.

[72] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2004.

[73] G.-M. Chiu, “The Odd-Even Turn Model for Adaptive Routing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 7, pp. 729–738, July 2000.

[74] J. Hu and R. Marculescu, “DyAD - Smart Routing for Networks-on-Chip,” in *Proc. 41st Design Automation Conference*, 2004, pp. 260–263.

[75] Y. H. Song and T. M. Pinkston, “A Progressive Approach to Handling Message-Dependent Deadlock in Parallel Computer Systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 3, pp. 259–275, Mar. 2003.

[76] C. J. Glass and L. M. Ni, “The Turn Model for Adaptive Routing,” in *Proc. 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 278–287.

[77] M. K. F. Schafer, T. Hollstein, H. Zimmer, and M. Glesner, “Deadlock-Free Routing and Component Placement for Irregular Mesh-based Networks-on-Chip,” in *Proc. ICCAD-2005 Computer-Aided Design IEEE/ACM International Conference on*, Nov. 6–10, 2005, pp. 238–245.

[78] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, “Routing Table Minimization for Irregular Mesh NoCs,” in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007, pp. 942–947.

[79] A. Hansson, K. Goossens, and A. Radulescu, "UMARS: A Unified Approach to Mapping and Routing in a Combined Guaranteed Service and Best-Effort Network-on-Chip Architecture," Philips Research Technical Report 2005/00340, Tech. Rep., April 2005.

[80] Duato, J., "A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993.

[81] S. Taktak, J.-L. Desbarbieux, and E. Encrenaz, "A Tool for Automatic Detection of Deadlock in Wormhole Networks on Chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 1–22, 2008.

[82] Duato, J., "A Theory of Fault-Tolerant Routing in Wormhole Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 790–802, Aug. 1997.

[83] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems on Chips," in *Proc. IFIP International Conference on Very Large Scale Integration*, Oct. 16–18, 2006, pp. 158–163.

[84] K. Goossens, J. Dielissen, O. P. Gangwal, S. G. Pestana, A. Radulescu, and E. Rijpkema, "A design flow for application-specific networks on chip with guaranteed performance to accelerate soc design and verification," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005, pp. 1182–1187.

[85] K. G. Andreas Hansson and A. Rădulescu, "Avoiding Message-Dependent Deadlock in Network-Based Systems on Chip," *VLSI Design*, 2007, http://repository.tudelft.nl/file/1225531/382523.

[86] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Design Test of Computers*, vol. 22, no. 5, pp. 414–421, Sept. 2005.

[87] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS Architecture and Design Process for Network on Chip," *Journal of System Architecture*, vol. 50, pp. 105–128, 2004.

[88] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for

Networks on Chip," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 350–355.

[89] T. Bjerregaard and J. Sparso, "A Scheduling Discipline for Latency and Bandwidth Guarantees in Asynchronous Network-on-Chip," in *Proc. 11th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2005*, Mar. 14–16, 2005, pp. 34–43.

[90] D. Siguenza-Tortosa and J. Nurmi, "Proteo: A New Approach to Network-on-Chip," Malaga (Spain), Tech. Rep., 2002.

[91] T. Bjerregaard and J. Sparso, "A Router Architecture for Connection-Oriented Service Guarantees in the. MANGO Clockless Network-on-Chip," in *Proc. Design, Automation and Test in Europe*, 2005, pp. 1226–1231.

[92] M. D. Harmanci, N. P. Escudero, Y. Leblebici, and P. Ienne, "Providing QoS to Connection-less Packet-switched NoC by Implementing DiffServ Functionalities," in *Proc. International Symposium on System-on-Chip*, Nov. 16–18, 2004, pp. 37–40.

[93] D. Castells-Rufas, J. Joven, and J. Carrabina, "A Validation And Performance Evaluation Tool for ProtoNoC," in *Proc. International Symposium on System-on-Chip*, Nov. 13–16, 2006, pp. 1–4.

[94] M. Coenen, S. Murali, A. Radulescu, K. Goossens, and G. De Micheli, "A Buffer-Sizing Algorithm for Networks on Chip Using TDMA and Credit-Based End-to-End Flow Control," in *Proc. 4th international conference Hardware/software codesign and system synthesis CODES+ISSS '06*, Oct. 22–25, 2006, pp. 130–135.

[95] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A Network on Chip Architecture and Design Methodology," in *Proc. IEEE Computer Society Annual Symposium on VLSI*, Apr. 25–26, 2002, pp. 105–112.

[96] J. Hu and R. Marculescu, "Application-Specific Buffer Space Allocation for Networks-on-Chip Router Design," in *Proc. ICCAD-2004 Computer Aided Design IEEE/ACM International Conference on*, Nov. 7–11, 2004, pp. 354–361.

[97] I. Saastamoinen, M. Alho, and J. Nurmi, "Buffer Implementation for Proteo Network-on-Chip," in *Proc. International Symposium on Circuits and Systems ISCAS '03*, vol. 2, May 25–28, 2003, pp. II–113–II–116.

[98] R. Mullins, A. West, and S. Moore, “Low-Latency Virtual-Channel Routers for On-Chip Networks,” in *Proc. 31st Annual International Symposium on Computer Architecture*, June 19–23, 2004, pp. 188–197.

[99] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, “Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC,” in *Proc. th Symposium on Integrated Circuits and Systems Design*, Sept. 4–7, 2005, pp. 178–183.

[100] C. A. Zeferino and A. A. Susin, “SoCIN: A Parametric and Scalable Network-on-Chip,” in *Proc. 16th Symposium on Integrated Circuits and Systems Design SBCCI 2003*, Sept. 8–11, 2003, pp. 169–174.

[101] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, “The Nostrum backbone-a communication protocol stack for Networks on Chip,” in *Proc. 17th International Conference on VLSI Design*, 2004, pp. 693–696.

[102] M. Dehyadgari, M. Nickray, A. Afzali-kusha, and Z. Navabi, “A New Protocol Stack Model for Network on Chip,” in *Proc. IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, vol. 00, Mar. 2–3, 2006, p. 3pp.

[103] K. Goossens, J. van Meerbergen, A. Peeters, and R. Wielage, “Networks on Silicon: Combining Best-Effort and Guaranteed Services,” in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, Mar. 4–8, 2002, pp. 423–425.

[104] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, “Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design,” in *Proc. Design Automation Conference*, 2001, pp. 667–672.

[105] H. Zimmermann, “OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection,” *IEEE Trans. Commun.*, vol. 28, no. 4, pp. 425–432, Apr. 1980.

[106] A. Baghdadi, D. Lyonnard, N. Zergainoh, and A. Jerraya, “An Efficient Architecture Model for Systematic Design of Application-Specific. Multiprocessor SoC,” in *DATE ’01: Proceedings of the conference on Design, automation and test in Europe*, 2001, pp. 55–63.

[107] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, “Automatic Generation of Application-Specific Architectures for Heterogeneous

Multiprocessor System-on-Chip," in *DAC '01: Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 518–523.

[108] S. Kim and S. Ha, "Efficient Exploration of Bus-Based System-on-Chip Architectures," *IEEE Trans. VLSI Syst.*, vol. 14, no. 7, pp. 681–692, July 2006.

[109] "AMBA Designer," ARM Ltd., 2004, http://www.arm.com/products/system-ip/amba-design-tools/amba-designer.php.

[110] Arteris S.A. – The Network on Chip Company, 2005, http://www.arteris.com.

[111] iNoCs SaRL – Structured Interconnects, 2007, http://www.inocs.com.

[112] Silistix Ltd. – CHAINworks, 2006, http://www.silistix.com.

[113] Sonics Inc – On Chip Communication Network for Advanced SoCs, 2004, http://www.sonicsinc.com.

[114] Erno Salminen, Ari Kulmala, and Timo D. Hämäläinen, "Survey of Network-on-chip Proposals," Tech. Rep., 2008.

[115] J. Xu, W. Wolf, J. Henkel, S. Chakradhar, and T. Lv, "A Case Study in Networks-on-Chip Design for Embedded Video," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, Feb. 16–20, 2004, pp. 770–775.

[116] G. Fen, W. Ning, and W. Qi, "Simulation and Performance Evaluation for Network on Chip Design Using OPNET," in *Proc. TENCON 2007 - 2007 IEEE Region 10 Conference*, Oct. 2007, pp. 1–4.

[117] Y.-R. Sun, "Simulation and Evaluation for a Network on Chip Architecture Using NS-2," Master's thesis, 2002.

[118] R. Lemaire, F. Clermidy, Y. Durand, D. Lattard, and A. A. Jerraya, "Performance Evaluation of a NoC-Based Design for MC-CDMA Telecommunications Using NS-2," in *Proc. 16th IEEE International Workshop on Rapid System Prototyping (RSP 2005)*, June 8–10, 2005, pp. 24–30.

[119] J. Bainbridge and S. Furber, "Chain: a Delay-Insensitive Chip Area Interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, Sept. 2002.

[120] D. Wiklund and D. Liu, “SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems,” in *Proc. International Parallel and Distributed Processing Symposium*, Apr. 22–26, 2003, p. 8pp.

[121] J. Chan and S. Parameswaran, “NoCGEN: A Template Based Reuse Methodology for Networks on Chip Architecture,” in *Proc. 17th International Conference on VLSI Design*, 2004, pp. 717–720.

[122] F. Moraes, N. Calazans, A. Mello, L. Mller, and L. Ost, “HERMES: An Infrastructure for Low Area Overhead Packet-switching Networks on Chip,” *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69 – 93, 2004.

[123] L. Ost, A. Mello, J. Palma, F. Moraes, and N. Calazans, “MAIA A Framework for Networks on Chip Generation and Verification,” in *Proc. Asia and South Pacific Design Automation Conference the ASP-DAC 2005*, vol. 1, Jan. 18–21, 2005, pp. 49–52.

[124] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, “NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, Feb. 2005.

[125] S. Pasricha, N. Dutt, and M. Ben-Romdhane, “Using TLM for Exploring Bus-based SoC Communication architectures,” in *Proc. 16th IEEE International Conference on Application-Specific Systems, Architecture Processors ASAP 2005*, July 23–25, 2005, pp. 79–85.

[126] B. Talwar and B. Amrutur, “A System-C based Microarchitectural Exploration Framework for Latency, Power and Performance Trade-offs of On-Chip Interconnection Networks,” in *First International Workshoop on Network on Chip Architectures (NoCArc)*, 2008.

[127] H. Lebreton and P. Vivet, “Power Modeling in SystemC at Transaction Level, Application to a DVFS Architecture,” vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 463–466.

[128] U. Y. Ogras, J. Hu, and R. Marculescu, “Communication-Centric SoC Design for Nanoscale Domain,” Tech. Rep., July 23–25, 2005.

[129] J. Joven, D. Castells-Rufas, S. Risueo, E. Fernandez, and J. Carrabina, “NoCMaker & j2eMPI A Complete HW-SW Rapid Prototyping EDA Tool for Design Space Exploration of NoC-based MPSoCs,” in *IEEE/ACM Design, Automation and Test in Europe*, 2009.

[130] P. Bellows and B. Hutchings, "JHDL - An HDL for Reconfigurable Systems," in *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 15–17, 1998, pp. 175–184.

[131] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD Suite for High-Performance FPGA Design," in *Proc. Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '99*, Apr. 21–23, 1999, pp. 12–24.

[132] "Xilinx® Fast Simplex Link (FSL)," Xilinx, http://www.xilinx.com/products/ipcenter/FSL.htm.

[133] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, 2003.

[134] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures," *Computers, IEEE Transactions on*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[135] E. Bolotin, A. Morgenshtein, I. Cidon, R. Ginosar, and A. Kolodny, "Automatic Hardware-Efficient SoC Integration by QoS Network on Chip," in *Proc. 11th IEEE International Conference on Electronics, Circuits and Systems ICECS 2004*, Dec. 13–15, 2004, pp. 479–482.

[136] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., 2006.

[137] G. Palermo and C. Silvano, "PIRATE: A Framework for Power/Performance Exploration of Network-on-Chip Architectures," in *PATMOS*, 2004, pp. 521–531.

[138] J. Chan and S. Parameswaran, "NoCEE - Energy Macro-Model Extraction Methodology for Network on Chip Routers," in *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, 2005, pp. 254–259.

[139] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks," in *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, 2002, pp. 294–305.

[140] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, “ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration,” in *Proc. DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, Apr. 20–24, 2009, pp. 423–428.

[141] R. Nagpal, A. Madan, A. Bhardwaj, and Y. N. Srikant, “INTACTE: An Interconnect Area, Delay, and Energy Estimation Tool for Microarchitectural Explorations,” in *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, 2007, pp. 238–247.

[142] “The Message Passing Interface (MPI) standard,” http://www.mcs.anl.gov/mpi/.

[143] “The uClinux Embedded Linux/Microcontroller Project,” http://www.uclinux.org.

[144] “embedded Configurable operating system (eCos),” http://ecos.sourceware.org/.

[145] “Altera Quartus II,” Altera, http://www.altera.com.

[146] “Xilinx Design Flow – ISE Design Suite,” Xilinx, http://www.xilinx.com/tools/designtools.htm.

[147] S. Lukovic and L. Fiorin, “An Automated Design Flow for NoC-based MPSoCs on FPGA,” in *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. IEEE Computer Society, 2008, pp. 58–64.

[148] “SoC Encounter,” Cadence, http://www.cadence.com.

[149] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, “System Level Design: Orthogonalization of Concerns and Platform-Based Design,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[150] D. J. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, C. M. K. San Francisco, Ed. San Francisco, CA: Morgan Kaufmann, 1999.

[151] P. Steenkiste, “A High-Speed Network Interface for Distributed-Memory Systems: Architecture and Applications,” *ACM Trans. Comput. Syst.*, vol. 15, no. 1, pp. 75–109, 1997.

[152] W. S. Lee, W. Dally, S. Keckler, N. Carter, and A. Chang, "An Efficient, Protected Message Interface," *Computer*, vol. 31, no. 11, pp. 69–75, Nov 1998.

[153] T. Callahan and S. Goldstein, "NIFDY: A Low Overhead, High Throughput Network Interface," in *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, Jun 1995, pp. 230–241.

[154] A. Chien, M. Hill, and S. Mukherjee, "Design Challenges for High-Performance Network Interfaces," *Computer*, vol. 31, no. 11, pp. 42–44, Nov 1998.

[155] P. Bhojwani and R. Mahapatra, "Interfacing Cores with On-chip Packet-Switched Networks," in *Proc. 16th International Conference on VLSI Design*, Jan. 4–8, 2003, pp. 382–387.

[156] J. Liang, S. Swaminathan, and R. Tessier, "ASOC: a scalable, single-chip communications architecture," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, Oct. 15–19, 2000, pp. 37–46.

[157] J. Liu, L.-R. Zheng, and H. Tenhunen, "Interconnect Intellectual Property for Network-on-Chip (NoC)," *J. Syst. Archit.*, vol. 50, no. 2-3, pp. 65–79, 2004.

[158] P. L. Ville Rantala, Teijo Lehtonen and J. Plosila, "Dual NI Architectures for Fault Tolerant NoC," in *Proc. Design, Automation and Test in Europe DATE '09*, 2009.

[159] T. Bjerregaard, S. Mahadevan, R. G. Olsen, and J. Sparsoe, "An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip," in *Proc. International Symposium on System-on-Chip*, Nov. 17–17, 2005, pp. 171–174.

[160] T. Bjerregaard and J. SparsOE, "Packetizing OCP Transactions in the MANGO Network-on-Chip," in *Proc. 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools DSD 2006*, 2006, pp. 657–664.

[161] E. B. Jung, H. W. Cho, N. Park, and Y. H. Song, "Sona: An on-chip network for scalable interconnection of amba-based ips," in *International Conference on Computational Science (4)*, 2006, pp. 244–251.

[162] S.-H. Hsu, Y.-X. Lin, and J.-M. Jou, “Design of a dual-mode noc router integrated with network interface for amba-based ips,” nov. 2006, pp. 211 –214.

[163] P. Martin, “Design of a Virtual Component Neutral Network-on-Chip Transaction Layer,” in *Proc. Design, Automation and Test in Europe*, 2005, pp. 336–337.

[164] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, “An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration,” *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 24, no. 1, pp. 4–17, 2005.

[165] P. Semiconductors, “Device Transaction Level (DTL) Protocol Specification. Version 2.2,” July 2002.

[166] D. Ludovici, A. Strano, D. Bertozzi, L. Benini, and G. Gaydadjiev, “Comparing tightly and loosely coupled mesochronous synchronizers in a noc switch architecture,” in *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, May 2009, pp. 244–249.

[167] “Synplicity Synplify® Premier,” Synopsis, http://www.synopsis.com.

[168] W. J. Dally, “Virtual-Channel Flow Control,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 194–205, Mar. 1992.

[169] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, “A Methodology for Mapping Multiple Use-Cases onto Networks on Chips,” in *Proc. Design, Automation and Test in Europe DATE '06*, vol. 1, Mar. 6–10, 2006, pp. 1–6.

[170] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini, “Bringing NoCs to 65 nm,” *IEEE Micro*, vol. 27, no. 5, pp. 75–85, Sept. 2007.

[171] T. Marescaux and H. Corporaal, “Introducing the SuperGT Network-on-Chip; SuperGT QoS: more than just GT,” in *Proc. 44th ACM/IEEE Design Automation Conference DAC '07*, June 4–8, 2007, pp. 116–121.

[172] D. Andreasson and S. Kumar, "On Improving Best-Effort throughput by better utilization of Guaranteed-Throughput channels in an on-chip communication system," in *Proceeding of 22th IEEE Norchip Conference*, 2004.

[173] ——, "Improving BE traffic QoS using GT slack in NoC systems," in *Proceeding of 23th IEEE Norchip Conference*, vol. 23, 2005, pp. 44–47.

[174] A. Hansson and K. Goossens, "Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases," in *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, 2007, pp. 233–242.

[175] E. Carara, N. Calazans, and F. Moraes, "Managing QoS Flows at Task Level in NoC-Based MPSoCs," in *proceeding VLSI-SoC09 (to appear)*, 2009.

[176] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, "Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis," *IET Computers Digital Techniques*, vol. 3, no. 5, pp. 398–412, Sept. 2009.

[177] A. Hansson, M. Subburaman, and K. Goossens, "Aelite: A Flit-Synchronous Network on Chip with Composable and Predictable Services," in *Proc. DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, Apr. 20–24, 2009, pp. 250–255.

[178] J. Wang, Y. Li, Q. Peng, and T. Tan, "A dynamic priority arbiter for Network-on-Chip," in *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, July 2009, pp. 253–256.

[179] M. Winter and G. P. Fettweis, "A Network-on-Chip Channel Allocator for Run-Time Task Scheduling in Multi-Processor System-on-Chips," in *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008, pp. 133–140.

[180] O. Moreira, J. J.-D. Mol, and M. Bekooij, "Online Resource Management in a Multiprocessor with a Network-on-Chip," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 1557–1564.

[181] Z. Lu and R. Haukilahti, "NOC Application Programming Interfaces: High Level Communication Primitives and Operating System Services for Power Management," pp. 239–260, 2003.

[182] "Real-Time Operating System for Multiprocessor Systems (RTEMS)," http://www.rtems.com.

[183] A. Marongiu and L. Benini, "Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy," in *Proc. DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, Apr. 20–24, 2009, pp. 809–814.

[184] "IEEE 754: Standard for Binary Floating-Point Arithmetic," 1985, http://grouper.ieee.org/groups/754/.

[185] "ARM Cortex-R4(F) – ARM Processor," ARM Ltd., 2006, http://www.arm.com/products/processors/cortex-r/cortex-r4.php.

[186] "ARM Cortex-M1 – ARM Processor," ARM Ltd., 2007, http://www.arm.com/products/processors/cortex-m/cortex-m1.php.

[187] D. Kreindler, "How to implement double-precision floating-point on FPGAs," Altera, Tech. Rep., 2007, http://www.pldesignline.com/202200714?printableArticle=true.

[188] D. Strenski, "FPGA Floating Point Performance – A Pencil and Paper Evaluation," Cray Inc., Tech. Rep., 2007, http://www.hpcwire.com/features/FPGA_Floating_Point_Performance.html.

[189] "Altera Floating Point Megafunctions," Altera, http://www.altera.com/products/ip/dsp/arithmetic/m-alt-float-point.html.

[190] "Virtex-5 APU Floating-Point Unit v1.01a," Xilinx, http://www.xilinx.com/support/documentation/ip_documentation/apu_fpu_virtex5.pdf.

[191] "Floating-Point Operator v5.0," Xilinx, http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

[192] "TIE Application Notes and Examples," ARM Ltd., 2006, http://www.tensilica.com/products/literature-docs/application-notes/tie-application-notes.htm.

[193] G. Ezer, "Xtensa with User Defined DSP Coprocessor Microarchitectures," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 335–342.

[194] "Cortex™-R4 and Cortex-R4F Technical Reference Manual (Revision: r1p3)," Tech. Rep., 2009, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0363e/index.html.

[195] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, no. 9, pp. 948–960, Sept. 1972.

[196] "PVM (Parallel Virtual Machine)," http://www.csm.ornl.gov/pvm/.

[197] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[198] A. S. Leon, J. L. Shin, K. W. Tam, W. Bryg, F. Schumacher, P. Kongetira, D. Weisner, and A. Strong, "A Power-Efficient High-Throughput 32-Thread SPARC Processor," in *Proc. Digest of Technical Papers. IEEE International Solid-State Circuits Conference ISSCC 2006*, Feb. 6–9, 2006, pp. 295–304.

[199] U. M. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park, "An 8-Core 64-Thread 64b Power-Efficient SPARC SoC," in *Proc. Digest of Technical Papers. IEEE International Solid-State Circuits Conference ISSCC 2007*, Feb. 11–15, 2007, pp. 108–590.

[200] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[201] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[202] W. Gropp and E. Lusk, "The MPI Communication Library: its Design and a Portable Implementation," in *Proc. Scalable Parallel Libraries Conference*, Oct. 6–8, 1993, pp. 160–165.

[203] D. W. Walker and J. J. Dongarra, "MPI: A Standard Message Passing Interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[204] "Open MPI: Open Source High Performance Computing," http://www.open-mpi.org/.

[205] "The OpenMP API Specification for Parallel Programming," http://www.openmp.org/.

[206] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support For Lock-free Data Structures," in *Proc. 20th Annual International Symposium on Computer Architecture*, May 16–19, 1993, pp. 289–300.

[207] T. Harris and K. Fraser, "Language support for lightweight transactions," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003, pp. 388–402.

[208] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Proc. 31st Annual International Symposium on Computer Architecture*, June 19–23, 2004, pp. 102–113.

[209] A. McDonald, B. D. Carlstrom, J. Chung, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Transactional Memory: The Hardware-Software Interface," *IEEE Micro*, vol. 27, no. 1, pp. 67–76, Jan. 2007.

[210] B. Lewis and D. J. Berg, *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall Press, 1995.

[211] V. Pillet, J. Labarta, T. Corts, and S. Girona, "PARAVER: A Tool to Visualize and Analyse Parallel Code," in *In Proceedings of WoTUG-18: Transputer and occam Developments, volume 44 of Transputer and Occam Engineering*, 1995, pp. 17–31.

[212] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[213] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," pp. 751–760, 2001.

[214] S. Vinoski, "New Features for CORBA 3.0," *Commun. ACM*, vol. 41, no. 10, pp. 44–52, 1998.

[215] W. T. Michal, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, “StreamIt: A Compiler for Streaming Applications,” Tech. Rep., 2001.

[216] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, “Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management,” in *Proc. International Conference on Hardware/Software Codesign and System Synthesis CODES + ISSS 2004*, 2004, pp. 48–53.

[217] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. O’Reilly, 1998.

[218] F. Rincón, F. Moya, J. Barba, D. Villa, F. J. Villanueva, and J. C. López, “A New Model for NoC-based Distributed Heterogeneous System Design,” in *PARCO’05: International Conference on Parallel Computing*, 2005, pp. 777–784.

[219] C. Ferri, T. Moreshet, R. I. Bahar, L. Benini, and M. Herlihy, “A Hardware/Software Framework for Supporting Transactional Memory in a MPSoC Environment,” vol. 35, no. 1, 2007, pp. 47–54.

[220] “ARM11 MPCore Processors,” ARM Ltd, 2004, http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php.

[221] Q. Meunier and F. Petrot, “Lightweight Transactional Memory systems for large scale shared memory MPSoCs,” in *Proc. Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference NEWCAS-TAISA ’09*, June 2009, pp. 1–4.

[222] M. Forsell, “A Scalable High-Performance Computing Solution for Networks on Chips,” *IEEE Micro*, vol. 22, no. 5, pp. 46–55, Sept. 2002.

[223] W.-C. Jeun and S. Ha, “Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS,” in *Proc. Asia and South Pacific Design Automation Conference ASP-DAC ’07*, Jan. 23–26, 2007, pp. 44–49.

[224] K. OBrien, K. OBrien, Z. Sura, T. Chen, and T. Zhang, “Supporting OpenMP on Cell,” *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.

[225] F. Liu and V. Chaudhary, “Extending OpenMP for Heterogeneous Chip Multiprocessors,” in *Proc. International Conference on Parallel Processing*, 2003, pp. 161–168.

[226] A. A. Jerraya, A. Bouchhima, and F. Pétrot, “Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC,” in *DAC '06: Proceedings of the 43rd annual Design Automation Conference*. New York, NY, USA: ACM, 2006, pp. 280–285.

[227] L. Kriaa, A. Bouchhima, M. Gligor, A. Fouillart, F. Pétrot, and A. Jerraya, “Parallel Programming of Multi-processor SoC: A HW-SW Interface Perspective,” *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 68–92, 2008.

[228] M. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. Jerraya, “Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study,” in *DAC'04: Proceedings of the 41st Design Automation Conference, 2004*, 2004, pp. 908–913.

[229] M. Bonaciu, S. Bouchhima, W. Youssef, X. Chen, W. Cesario, and A. Jerraya, “High-Level Architecture Exploration for MPEG4 Encoder with Custom Parameters,” in *Proc. Asia and South Pacific Conference on Design Automation*, Jan. 24–27, 2006, p. 6pp.

[230] P. Francesco, P. Antonio, and P. Marchal, “Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture,” in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, 2005, pp. 736–741.

[231] A. Dalla Torre, M. Ruggiero, and L. Benini, “MP-Queue: An Efficient Communication Library for Embedded Streaming Multimedia Platforms,” in *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, Oct. 2007, pp. 105–110.

[232] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, “MPI Microtask for Programming the Cell Broadband Engine™ Processor,” *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, 2006.

[233] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann, “A Reconfigurable Cluster-on-Chip Architecture with MPI Communication Layer,” in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 350–352.

[234] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda, “SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips,” in *Proc. International Conference on Reconfigurable Computing and FPGAs ReConFig '08*, Dec. 3–5, 2008, pp. 187–192.

[235] J. Psota and A. Agarwal, “rMPI: Message Passing on Multicore Processors with on-chip Interconnect,” *Lecture Notes in Computer Science*, vol. 4917, p. 22, 2008.

[236] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, “The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs,” *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[237] M. Saldaña and P. Chow, “TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs,” in *Field Programmable Logic and Applications, 2006. FPL ’06. International Conference on*, Aug. 2006, pp. 1–6.

[238] A. Agbaria, D.-I. Kang, and K. Singh, “LMPI: MPI for Heterogeneous Embedded Distributed Systems,” in *Proc. 12th International Conference on Parallel and Distributed Systems ICPADS 2006*, vol. 1, 2006, p. 8pp.

[239] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, and M. Robert, “MPI-Based Adaptive Task Migration Support on the HS-Scale System,” *VLSI, IEEE Computer Society Annual Symposium on*, vol. 0, pp. 105–110, 2008.

[240] G. M. Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, and M. Robert, “An Adaptive Message Passing MPSoC Framework,” *International Journal of Reconfigurable Computing*, 2009.

[241] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, “HS-Scale: a Hardware-Software Scalable MP-SoC Architecture for Embedded Systems,” in *VLSI, 2007. ISVLSI ’07. IEEE Computer Society Annual Symposium on*, March 2007, pp. 21–28.

[242] “The Multicore Association – The Multicore Communication API (MCAPI),” http://www.multicore-association.org/home.php.

[243] T. P. McMahon and A. Skjellum, “eMPI/eMPICH: embedding MPI,” in *Proc. Second MPI Developer’s Conference*, July 1–2, 1996, pp. 180–184.

[244] Centre de Prototips i Solucions Hardware-Software Lab, “NoCMaker Project,” 2007, http://www.cephis.uab.es/proj/public/nocmaker/index.xhtml.

[245] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. the MIT Press, 1999.

[246] R. Brightwell and K. D. Underwood, "Evaluation of an Eager Protocol Optimization for MPI," in *PVM/MPI*, 2003, pp. 327–334.

[247] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi, and M. Poncino, "Energy-Efficient Multiprocessor Systems-on-Chip for Embedded Computing: Exploring Programming Models and Their Architectural Support," *Computers, IEEE Transactions on*, vol. 56, no. 5, pp. 606–621, May 2007.

[248] P. Mucci and K. London, "The MPBench Report," Tech. Rep., 1998.

[249] D. Grove and P. Coddington, "Precise MPI Performance Measurement Using MPIBench," in *In Proceedings of HPC Asia*, 2001.

[250] O. Villa, G. Palermo, and C. Silvano, "Efficiency and Scalability of Barrier Synchronization on NoC Based Many-Core Architectures," in *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008, pp. 81–90.

[251] A. Dorta, C. Rodriguez, and F. de Sande, "The OpenMP Source Code Repository," in *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, Feb. 2005, pp. 244–250.

[252] J. Joven *et al.*, "Application-to-Packets QoS Support - A Vertical Integrated Approach for NoC-based MPSoC," 2010.

[253] J. Joven, D. Castells-Rufas, and J. Carrabina, "An efficient MPI Microtask Communication Stack for NoC-based MPSoC Architectures," in *Fourth Computer Architecture and Compilation for Embedded Systems*, 2008.

[254] J. Joven, "A LightweightMPI-based ProgrammingModel and its HW Support for NoC-based MPSoCs," in *IEEE/ACM Design, Automation and Test in Europe (PhD Forum)*, 2009.

[255] M. Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, Sept. 2007.

[256] M. Butts, A. M. Jones, and P. Wasson, “A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing,” in *Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM 2007*, Apr. 23–25, 2007, pp. 55–64.

[257] “Tilera Corporation,” Tilera, http://www.tilera.com.

[258] “80-Core Programmable Teraflops Research Processor Chip,” Intel, http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[259] “Tera-scale Computing Research Program,” Intel, http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[260] “Single-chip Cloud Computing,” Intel, http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[261] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, “Larrabee: A Many-Core x86 Architecture for Visual Computing,” *IEEE Micro*, vol. 29, no. 1, pp. 10–21, Jan. 2009.