Universitat Autònoma de Barcelona

Offloading Techniques to Improve Performance on MPI Applications in NoC-Based MPSoCs

Ph.D. Thesis Dissertation

Autor:

Eduard Fernandez Alonso

Director:

Jordi Carrabina i Bordoll

Co-Director:

Jaume Joven Murillo

Universitat Autònoma de Barcelona

Abril 2014

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.

Dr. Jordi Carrabina I Bordoll

Dr. Jaume Joven I Murillo

This work was carried out at Universitat Autònoma de Barcelona, and Recore Systems, Enschede, The Netherlands.

ABSTRACT

Future embedded System-on-Chip (SoC) will probably be made up of tens or hundreds of heterogeneous Intellectual Properties (IP) cores, which will execute one parallel application or even several applications running in parallel. These systems could be possible due to the constant evolution in technology that follows the Moore's law, which will lead us to integrate more transistors on a single dice, or the same number of transistors in a smaller dice. In embedded MPSoC systems, NoCs can provide a flexible communication infrastructure, in which several components such as microprocessor cores, MCU, DSP, GPU, memories and other IP components can be interconnected.

In this thesis, firstly, we present a complete development process created for developing MPSoCs on reconfigurable clusters by complementing the current SoC development process with additional steps to support parallel programming and software optimization. This work explains systematically problems and solutions to achieve a FPGA-based MPSoC following our systematic flow and offering tools and techniques to develop parallel applications for such systems.

Additionally, we show several programming models for embedded MPSoCs and propose the adoption of MPI for such systems and show some implementations created in this thesis over shared and distributed memory architectures.

Finally, the focus will be set on the overhead produced by MPI library and on trying to find solutions to minimize this overhead and then be able to accelerate the execution of the application, offloading some parts of the software stack to the Network Interface Controller.

RESUM

Probablement, el sistema-en-xip encastat futur estarà compost per desenes o centenars de nuclis de Propietat Intel·lectual heterogenis que executaran una aplicació paral·lela o fins i tot diverses aplicacions que funcionin en paral·lel. Aquests sistemes seran possible gràcies a l'evolució constant de la tecnologia que segueix la llei de Moore, que ens durà a integrar més transistors en un únic dau, o el mateix nombre de transistors en un dau més petit. En els sistemes MPSoC encastats, les xarxes intenrades (NoC) poden proporcionar una infraestructura de comunicació flexible, en què diversos components, com ara els nuclis microprocessadors, MCU, DSP, GPU, memòries i altres components IP, poden estar interconnectats.

En primer lloc, en aquesta tesi presentem un procés de desenvolupament complet creat per desenvolupar MPSoC en clústers reconfigurables tot complementant el procés de desenvolupament SoC actual amb passos addicionals per admetre la programació paral·lela i l'optimització del software. Aquest treball explica de manera sistemàtica els problemes i les solucions per aconseguir un MPSoC basat en FPGA seguint el nostre flux sistemàtic, i s'ofereixen eines i tècniques per desenvolupar aplicacions paral·leles per a aquests sistemes.

D'altra banda, descrivim diversos models de programació per a MPSoC encastats i proposem adoptar MPI per a aquests sistemes, i mostrem algunes implementacions creades en aquesta tesi amb arquitectures de memòria compartida i distribuïda. Finalment, ens centrem en la sobrecarrega de temps que produeix la llibreria MPI i intentarem trobar solucions per tal de minimitzar aquesta sobrecàrrega i, per tant, poder accelerar l'execució de l'aplicació, descarregant algunes parts del software stack al controlador d'interfície de la xarxa.

Time, what is time?

Time, so mortal concept, so human concept, so engineer concept. Without a clock the designer is lost, but the human is happy.

Time is life. When the clock strikes, a bit of your life moves away. Make it worth! Have a plan! Be one of the final five!

Some Peter Pan said once: "Life is a map, but it is quite confusing" Enjoy it then! Do not worry about the compass! Just be sure to do what you love to do and to be with who you want to be! ...And ignore those who want to make you waste your time.

Research.

ACKNOWLEDGEMENTS

I do not want to end the work without thanking all the people who had a certain impact on the evolution of the thesis or my evolution while doing the thesis. I want to thank all of them, even if this impact was a totally fantastic idea or advise, or a complete deception that was useful for nothing and absolutely avoidable.

However, first of all, I want to thank my thesis directors. Jaume Joven, who already guided me when I was doing the final computer science degree's project, and Jordi Carrabina, who is at the same time my boss with who I have been working for so long.

I want to thank so many people from CAIAC/Cephis group... that it is impossible to mention them all in these lines. However, ALL of them deserve the opportunity to walk the same path that I walked to obtain their own PhD as I had.

I want to specially thank my lab mates. First of all, David Castells, from who I've learned plenty of things. I really appreciate that. And Albert Saa, who I share the love for board games with.

Some other lab mates are no more with us (we like to say that they are living a better life now), but I want to thank them, specially Chak Ma... man, you still have my umbrella!

Although we are not at the same lab, I want to thank deeply Eloi Ramon and Carme Martinez for sharing knowledge, articles, breakfast, lunch time, dinner, funny moments... and really sad moments. Thank you.

I want to thank also my Dutch friends. Gerard for being so kind and let me work in his company; Kim for his guide while I was working in The Netherlands and for appreciating the "art" created with the Xentiums; Simon-Thais for taking care of me when I had some "minor" health problems (I must hug you, man!); Inès for being absolutely lovely!, Lois for being always smiling and sharing with us the nice adventure of living at Enschede, and Jordy, John, Alejandro, Sébastien,... and the rest of the crew.

I cannot forget Jarkko. Thank you for helping us to feel like in our own country, for your attention, for being always in a good mood, for your friendship. Moltes gràcies per tot. Aquest missatge és especial per a tu: "setze jutges d'un jutjat mengen fetge d'un penjat".

I leave for the end my family and friends. I want to thank my parents, who gave me everything that made possible this PhD. I want to thank my little sister for being just like she is... thank you! I'm always learning from you. I'm so proud of you. I want to thank specially my cousin Sergi for SO many happy moments we have lived together... and I hope we'll share many more.

I want to thank my friends, Albert, Cris, Adam (I cannot forget him...his parents will kill me), Alex, the D&D group (Miki, Xavi, Fer, Iza, Pepo), the Vampire group (Montse, Laia, Jorge, Sergi, Mac, Xavi, Pepo again),... the list is too extensive.

Finally, I leave for the end my more deep-rooted thanks to my little family, Laia and Curial. Thank you for everything. YOU are my time, YOU are my passion, YOU are my life, I definitively could not have made it without you. I love you both.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

A-0	Arithmetic Language version 0
ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
ALUT	Adaptive Look-up table
AMBA	Advanced Microcontroller Bus Architecture
ANoC	Asynchronous NoC
ANSI	American National Standards Institute
API	Application Programming Interface
APP	Application
ASIC	Application Specific Integrated Circuit
ccNUMA	cache coherence Non-Uniform Memory Access
CFG	Control Flow Graph
COMA	Cache Only Memory Access
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFG	Data Flow Graph
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DSP	Digital Signal Processor
FFT	Fast Fourier Transfor
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPR	General Purpose Register
GPU	Graphics Processor Unit
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HPC	High-Performance Computing

HW	Hardware
I/O	Input/Ouput
IC	Integrated Circuit
ID	Identification
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction-Level Parallelism
IP	Intellectual Properties
ISA	Instruction Set Architecture
JTAG	Join Test Action Group
LAN	Local Area Network
LCD	Liquid crystal display
LE	Logic Element
MCU	Microcontroller
MIMD	Multiple-Instruction Multiple-Data
MISD	Multiple-Instruction Single-Data
MM	Matrix Multiplication
MOS	Metal-Oxide-Semiconductor
MP	Message Passing
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
NA	Network Adapter
NI	Network Interface
NIC	Network Interface Controller
NoC	Network-on-Chip
NORMA	No Remote Memory Access
NUMA	Non-Uniform Memory Access
ocMPI	on-chip Message Passing Interface
OCP	Open Core Protocol
OCP-IP	Open Core Protocol-International Partnership
OpenCL	Open Computing Language

OpenMP	Open Multi-Processing
OpenMPI	Open Message Passing Interface
OSI	Open Systems Interconnection
OTF	Open Trace Format
PC	Personal computer
PCB	Printed Circuit Board
PE	Processor Element
PIM	Processor-in-Memory
PIO	Programmed Input/Output
PPP	Parallel Programming Pattern
QoS	Quality of services
RAM	Random Access Memory
RAMPSoC	Runtime Adaptive Multi-Processor System-on- Chip
RAMPSoC-MPI	RAMPSoC-Message Passing Interface
RISC	Reduced Instruction Set Computer
RNI	Resource Network Interface
ROM	Read-Only-Memory
SCC	Single Chip Cloud
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single-Instruction Multiple-Data
SiP	System-in-Package
SISD	Single-Instruction Single-Data
SoC	System-on-Chip
SOPC	System on a Programmable Chip
SRAM	Static Random Access Memory
SW	Software
TBB	Threading Building Blocks
TLP	Thread-Level Parallelism
TX	Transmission
UART	Universal Asynchronous Receiver/Transmitter

UMA	Uniform Memory Access
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1. Introduction

Ever since Lucy decided to start walking over her back legs, it is a human condition that each and every parent wishes the best for their descendents, wishes them to be more than themselves – more beautiful, more intelligent, more efficient. The same applies for engineers: they wish their little creatures to be "more", and our baby here is multiprocessors on-chip.

Due to the evolution of technology, future embedded System-on-Chip (SoC) processors will probably be made up of tens or hundreds of heterogeneous Intellectual Properties (IP) cores, which will execute one parallel application or even several applications running in parallel. Figure 1 shows the evolution (and prediction) of processor elements through time.

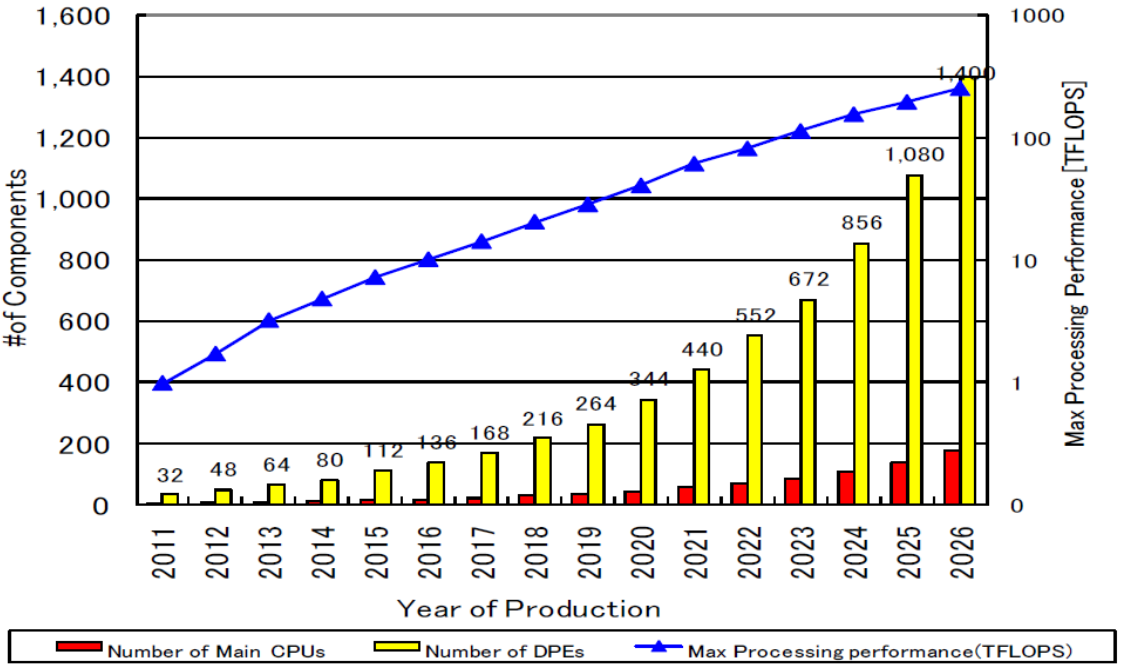


Figure 1 Number of devices versus time. [11]

Gordon Moore postulated in 1976 [1] that every 24 months approximately the number of transistors that can be integrated inside the same dice is multiplied by 2.

With such prediction, the genetic raw material is guaranteed to create every 24 months a new generation of powerful multiprocessors. But is this enough? Is it enough to face the challenges of upcoming applications?

We know now that the answer is NO: the mere evolution of technology is not enough, and it is not a sufficient condition to get better devices. Designers must adapt, find, change, and create new methodologies and electronic design automation tools to face the opportunities opened by technology. And yet, that is not enough. It is necessary to offer tools to the user to program such devices in an efficient way.

Across the last decade, system designers have been increasing their research efforts to develop an optimum interconnection system for multiprocessors devices.

This present thesis wants to contribute to the evolution of this world to achieve the goal of making our sons better.

The introduction is divided in two. The first part will talk about the history of the multiprocessor system-on-chip: when, how, and why of the network-on-chip concept. In the second part of the introduction, the highlight will be focused on the contributions of the thesis.

The rest of the thesis is divided into the following chapters:

- Chapter 2. General theory about MPSoCs architectures and interconnection systems, and general theory of parallel programming models for MPSoCs.
- Chapter 3: Presentation of a simulation environment and the soft-core-NoC-Based MPSoC system implementation used to support the of f-loading techniques shown in chapter 4.
- Chapter 3. Several MPI implementations developed in this thesis for NoC-based embedded systems using shared and parallel architectures.

- Chapter 4. Delivery protocol offloading design; Bus master offloading design; Synchronization operation offloading design.
- Chapter 5. Contributions and conclusions of the thesis.

1.1. Part 1– From computation to communication

Nowadays, the large computation capacity available to integrate in a single chip makes the complexity of management of such resources high, and, therefore, the focus of research is moved towards the communication system. In this part of the introduction, we are trying to give a brief in the evolution of embedded processors and computer architecture. First of all, we must say that there is plenty of literature talking about the history and evolution of computers, and it is not our intention to create a new reference document. However, as we mentioned above, we will give a brief summary using 60 milestones of the evolution through time of computers, extracted from [2][3][4][5][6][7][8][9][10]

1946: Eniac (Electronic Numerical Integrator and Computer) is born.

The first electronic "programmable" (general purpose) computer comes to life.

1948: Invention of the European transistor.

Herbert Mataré & Heinrich Welker independently create a germanium point-contact transistor in France.

1950: Assembly language for programming computers.

Low level programming language is introduced.

1952: First compiler for a programming language developed.

Grace Murray Hopper develops the first compiler for the A-0 (Arithmetic Language version 0) programming language.

1954: Silicon Transistors Offer Superior Operating Characteristics.

Morris Tanenbaum manufactures the first silicon transistor at Bell Labs, but Texas Instruments' engineers build and market the first commercial devices.

1957: First complete compiler developed.

IBM introduces the first complete compiler for FORTRAN.

1958: All semiconductor "Solid Circuit" is proved.

Jack Kilby produces a microcircuit with both active and passive components manufactured from a semiconductor material.

1959: Practical Monolithic Integrated Circuit Concept Patented.

Robert Noyce builds on Jean Hoerni's planar process to patent a monolithic integrated circuit structure that can be manufactured in high volume.

1961: Silicon Transistor Exceeds Germanium Speed.

Computer architect Seymour Cray funds development of the first silicon device to meet the performance demands of the world's fastest machine.

1963: Complementary MOS Circuit Configuration is invented.

Frank Wanlass invents the lowest power logic configuration but performance limitations impede early acceptance of today's dominant manufacturing technology.

1963: Institute of Electrical and Electronics Engineers.

IEEE is founded with the aim of advancing innovation and technological excellence for the benefit of humanity.

1964: First Commercial MOS IC introduced.

General Microelectronics uses a Metal-Oxide-Semiconductor (MOS) process to pack more transistors on a chip than bipolar ICs and builds the first calculator chip set using technology.

1965: Semiconductor Read-Only-Memory Chips Appear.

Semiconductor read-only-memories (ROMs) offer high density and low cost per bit.

1965: "Moore's Law" Predicts the Future of Integrated Circuits.

Fairchild's Director of R & D predicts the rate of increase of transistor density on an integrated circuit and establishes a yardstick for technology progress. Gordon Moore's article, published in April 1965 Electronics Magazine, establishes Moore's Law.

1966: Computer Aided Design Tools Developed for ICs.

IBM engineers pioneer computer-aided electronic design automation tools for reducing errors and speeding design time.

1966: Flynn's Taxonomy presented.

Michael J. Flynn proposes a classification of computer architecture according to global control and data and control flows.

1970: Programmable logic array developed.

Texas Instruments developes PLA, programmable logic arrays based on IDM's read-only memory.

1972: Bell Laboratories introduce C programming Language.

C programming language is originally developed to run on DEC PDP-11 with UNIX operating system.

1972: Cray Research, Inc. founded.

Seymour Cray founds Cray Research in Chippewa Falls, USA.

1974: Scaling of IC Process Design Rules Quantified.

IBM researcher Robert Dennard's paper on process scaling on MOS memories accelerates a global race to shrink physical dimensions and manufacture ever more complex integrated circuits.

1974: Digital Watch is First System-On-Chip Integrated Circuit.

The Microma liquid crystal display (LCD) digital watch is the first product to integrate a complete electronic system onto a single silicon chip, called a System-On-Chip or SOC.

1975: Microsoft appears.

Bill Gates and Paul Allen deliver a BASIC compiler to MITS, who agrees to distribute it as Altair BASIC.

1975: First Cray supercomputer.

Cray Research Inc. introduces first Cray-1 supercomputer.

1979: Single Chip Digital Signal Processor Introduced.

Bell Labs' single-chip DSP-1 Digital Signal Processor device architecture is optimized for electronic switching systems.

1980: First hard disk drive for microcomputers.

Seagate Technology creates ST506 hard disk with 5megabytes of data storage capacity.

1981: VHDL (VHSIC Hardware Description Language).

VHDL is initially proposed to comment the behavior of ASICs.

1983: Altera appears.

Founding of Altera Corporation by Hartmann, Magranet, Newhagen, and Sansbury.

1984: First reprogrammable device.

Altera introduces the first reprogrammable logic device. Previous devices could only be programmed once.

1984: Xilinx appears.

Founding of Xilinx company in Silicon Valley by Freeman, Vonderschmitt, and V Barnett II.

1984: First FPGA device presented.

Xilinx designs the first FPGA device, called XC2000 series.

1985: Join Test Action Group formed.

JTAG is formed to develop a methodology to test IC.

1985: First commercial RISC processor.

Acorn Computer Group develops first commercial RISC processor.

1985: Intel 80386 appears.

Intel 80386, or simply 386, comes to reality as a 32-bit microprocessor with 275,000 transistors (however, the fully functional version of 386 is introduced in 1986).

1987: Borland releases the first version of Turbo C.

Turbo C gives a huge boost to the use of C in embedded applications.

1989: Intel 80486 appears.

Intel 486 appears with 1,200,000 transistors. It is a 32-bit x86-pippelined-design microprocessor, with 8192-byte of SRAM memory and two separated 32-bit buses (data and address).

1990: JTAG becomes standard.

IEEE accepts JTAG as the standard 1149.1-1990.

1993: PowerPC appears.

IBM, Apple, and Motorola introduce PowerPC architecture.

1994: Message Passing Interface library.

MPI first version is developed by a group of parallel computer vendors, programmers and scientists.

1995: Java is released.

Sun introduces Java programming language with the philosophy of letting programmers write just one code and be able to run it on any platform (Write Once, Run Anywhere).

1996: Windows Embedded CE released.

Microsoft releases the first version of Windows Embedded CE.

1996: Advanced Microcontroller Bus Architecture appears.

ARM releases AMBA bus, which becomes the first de facto commercial standard bus.

1996: Virtual Socket Interface Alliance founded.

Several international companies from various segments of the SoC industry founded VSIA to enhance the productivity of the SoC design community.

1997: First Chip Multiprocessor architecture proposed.

By Hammond, Nayfeh and Olukotun from Standford University is presented including 8 cores.

1998: Open MultiProcessing language appears.

OpenMP is launched as an API for shared memory multiprocessing in C, C++ and Fortran.

1999: GPU is released.

Graphics Processing Unit processor is introduced by NVIDIA into graphical cards.

2000: Daytona DSP architecture MPSoC published.

First multiprocessor system-on-chip containing four processing elements interconnected by a bus is presented.

2001: Open Core Protocol International Partnership created.

Sonics, Nokia, Texas Instruments, MIPS, and UMC launch OCP-IP to Standardize IP Core Socket Interface.

2001: OCP 1.0 Bus released.

OCP-IP launches the first OCP Bus specifications.

2001: Networks-on-Chip new SoC paradigm.

Several academic authors claim for adoption of NoC as interconnection paradigm for SoCs.

2003: Will Networks-on-Chip Close the productivity gap?

Several problems related to busses appear as silicon technology continues advancing. Buses can efficiently connect up to 10 IP but do not scale to higher numbers.

2003: Arteris Company founded.

Arteris invents the industry's first commercial Network-on-Chip SoC interconnect IP solution.

2004: Tilera Company founded.

Tilera is founded as a manufacture of fables semiconductors and multicore embedded processor designs.

2005: First multicore commercial processors.

IBM, Intel, and AMD release their first multicore processors.

2005: CELL processor released.

IBM, Sony Computer Entertainment, and Toshiba Corporation release CELL multicore processor, which is part of PlayStation 3.

2007: Compute Unified Device Architecture SDK released.

NVIDIA introduces the initial CUDA SDK for Windows and Linux.

2007: Intel Polaris Processor.

Intel designs a multicore prototype with 80 cores called Polaris.

2008: Tilera TILE64 processor released.

Tilera Corporation launches TILE64 processor containing 64 programmable cores connected by a NoC.

2008: Intel's top processor i7 released.

Intel i7 general processor is released with 4 cores and 8 threads.

2008: VSI Alliance disappears

The VSI Alliance dissolved operations and transferred its ongoing work to other industry organizations

2010: Xilinx's Zynq FPGA families released.

Xilinx 7 FPGA families are launched with Zynq that has twin ARM Cortex-A9 cores.

2011: Altera releases QSYS system integration tool.

QSys is designed as the next generation SOPC builder tool.

2012: 22nm Manufacturing technology.

1.4 billion transistors into a single chip.

2014: To program multicore processors is still an uncompleted research.

As the number of transistors on a chip has been increasing through the IC history, processors have been progressing enhancing performance. Finally, this evolution allowed the creation of systems-on-chip where several IP components are integrated into a single dice. However, due also to evolution, busses showed scaling limitation when interconnecting several IP components on a SoC system. The solution appeared when the interconnection systems adopted the network-on-chip paradigm.

Over the last decade, academia and industry have been developing and enhancing networks-on-chip and trying to close the productivity gap (Figure 2).

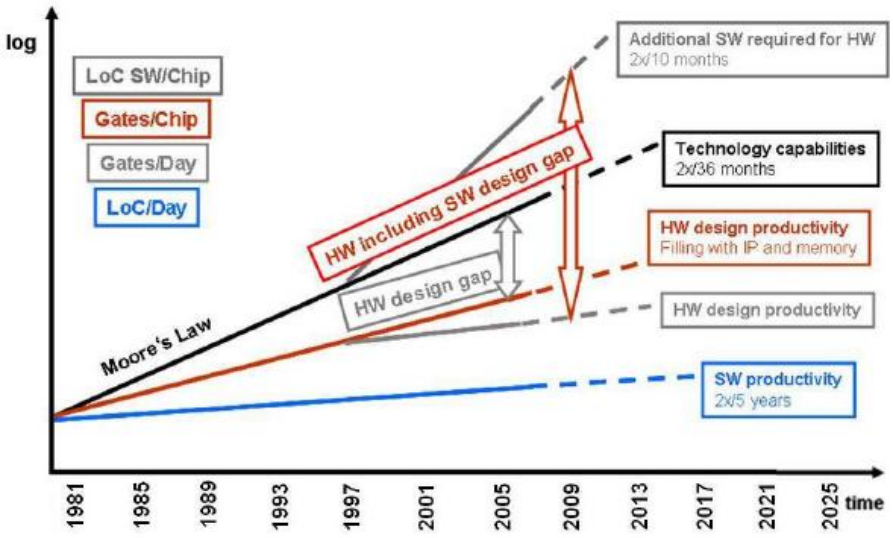


Figure 2 Hardware and software design gaps versus time [1]. [11]

[1] Source: ITRS 2011

1.2. Part 2– Objective of this thesis

The research scope of this thesis is the concept of NoC-based MPSoC architecture in close relation with programming models for such multicore systems. In particular, the goal of this thesis focuses is to propose a message passing interface API over MPSoCs with NoCs interconnection system that uses shared or distributed memory architectures and, also, to provide techniques to improve performance of the participant processors of the system, by off-loading software process to the network interface controller component.

Topics such as memory architectures, network interface controller design, programming model, and message passing interface are within the main scope of the thesis.

The general contribution of this thesis is to propose a general methodology to create MPSoC systems with networks-on-chip (Figure 3), to program such systems with a layered stack based on MPI-like API and techniques to improve the general performance of the system. The specific contributions aim to complete the existing methodologies to create, program and do performance analysis, and also, aim to help to improve the performance of the processors involved in the system by freeing them of doing some software process that will be off-loaded to the NoC, particularly to the network interface controller.

The specific tasks where the research is focussed to reach those objectives are:

- Develop a methodology to create MPSoCs including: (i) MPSoCs design, (ii) software stack to program MPSoCs, and (iii) performance analysis.

- Contribute to the NoC simulator tool from UAB.
- Develop a MPI library for embedded systems including implementations on: (i) distributed memory architectures systems (ocMPI), and (ii) shared memory architecture systems (STHROM platform and Recore's Multicore systems).
- Study of MPI library overheads.
- Design of new NIC facilities: (i) Bus Master NIC, (ii) Syncop NIC, and (iii) Rendez-vous NIC.

It must be pointed out that all contributions have been (completely or partially) implemented using FPGA platforms.

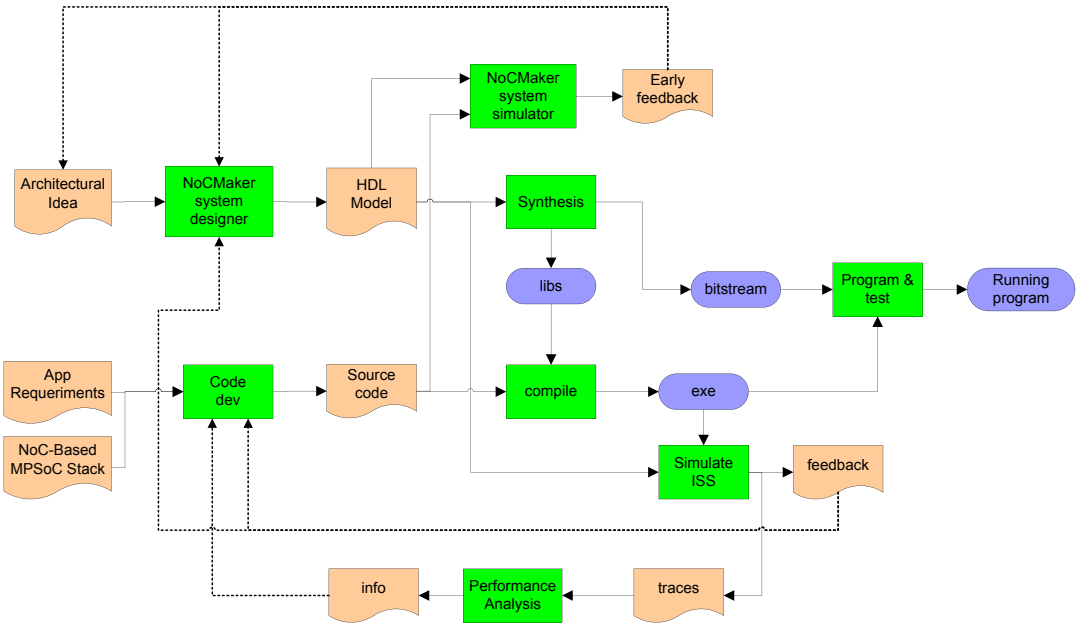


Figure 3 Full methodology flow

The contributions of the thesis are depicted in Figure 4, distributed according to their focus.

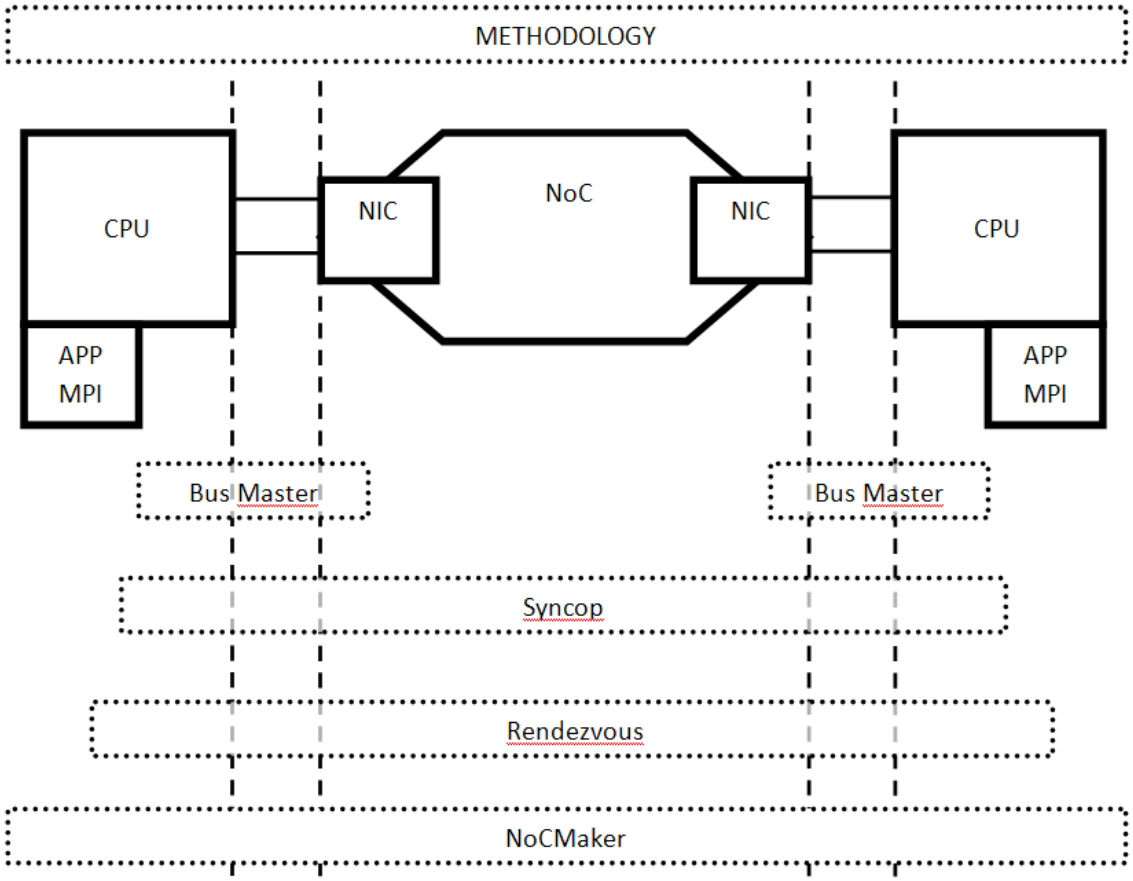


Figure 4 Thesis contribution graphical overview.

The work presented in this dissertation has been accomplished by the author of this thesis and several co-authors (academic and industrial researchers and engineers) together.

Finally, the complete list of papers on which this thesis is based on (with the full name of their co-authors) can be found in the appendix.

2. MPSoCs: theoretical background

2.1. Overview

To study MPSoCs is to talk about parallelism. So, we will start discussing a bit about parallelism before going deeper into MPSoCs systems. In this chapter, we will present a brief introduction on the current theoretical framework for parallelism.

2.2. Parallelism

Parallelism: Exploiting concurrency in a program with the goal of solving a problem in less time.

DR. Tim Mattson [2]

As this definition remarks, concurrency is crucial in parallelism and in parallel computing.

Concurrency: A property of a system in which multiple tasks that comprise the system remain active and make progress at the same time.

DR. Tim Mattson [2]

Concurrency is an old well-known term used in computer science theory. Carl Adam Petri was one of the first scientists to formalise and model concurrency with the Petri Nets. There are many other formalisms that model concurrency (actor model, the family model Process calculus, C-FSMs,…), but probably, Petri Nets is still the most famous model.

Nowadays, it is really common to find someone using a Smartphone with a dual-core or quad-core processor [12], or to buy a PC with 8 cores/threads [13], or even game consoles with 4 processor cores and 8 GPU cores [14]. It is obvious that parallelism and

[2] Principle engineer at Intel (Microprocessor Technology lab). [92]

parallel computing is the path to follow to create the upcoming devices and theoretical models. Parallelism has been used in several branches of technology and science, and the industrial requirements for solve complex problems make indispensable the use of parallelism.

Mankind is essentially social. We, humans, live in community, and it is in our nature to prone to parallelism to solve problems. It is unconceivable that one single person alone could construct highways or buildings, or design microchips. These human achievements are reached by a certain amount of people working together. These groups of people could have different qualities and skills, run different functions within the team, and so on.

In the computer science field, the solution of some problems turns again into parallelism. For some period of time, someone could wait for a new higher-performance-electronic-device to solve a certain problem. The continuous enhancing in technology made possible to increase the work clock frequency of transistors and, therefore, the new generations of electronic devices were faster and faster (in terms of clock frequency), and more dense in numbers of transistors.

Despite the benefits produced by this technological evolution, there was an important drawback: the increase in energy consumption and the need for power dissipation. As clock frequency rised, more switching from logic states were produced in the same amount of time and, therefore, energy consumption also raised. To compensate that, voltage supply has been decreasing for the core of the chips, down to the limit of the threshold voltage that cannot be lowered that much because the static energy consumption rises due to current at inverse biased PM junctions.

This is dynamic power consumption, which is still the major contribution to the chips (together with leakages) follows by the following formula:

$$P_{dynamic} = CV_{dd}^2 FA$$

where C depends on the wire , V_{DD} the supply voltage, F is the frequency, and A is the Activity (the amount of changes between 0 and 1 logic states per clock cycle). Additionally, the static power has been growing which each new technology node [15].

Both dynamic and static power consumption have also another incidence on the performance of the system due to the increase of the temperature in the device. Higher transistor density → More power density → Temperature rises → Performance lowers (speed decreases). Therefore, in many cases cooling is required. These issues could make the advances in technology shown in the Moore's law completely useless.

The exploit of parallelism helps to solve that situation, to control power consumption levels and power dissipation levels, while rises computing power. Figure 5 shows how parallelism helps in human challenges. In this case, the figure shows the evolution in time of the cost of human genome sequencing. It can be seen the falling in 2007 due to the use of parallel computers.

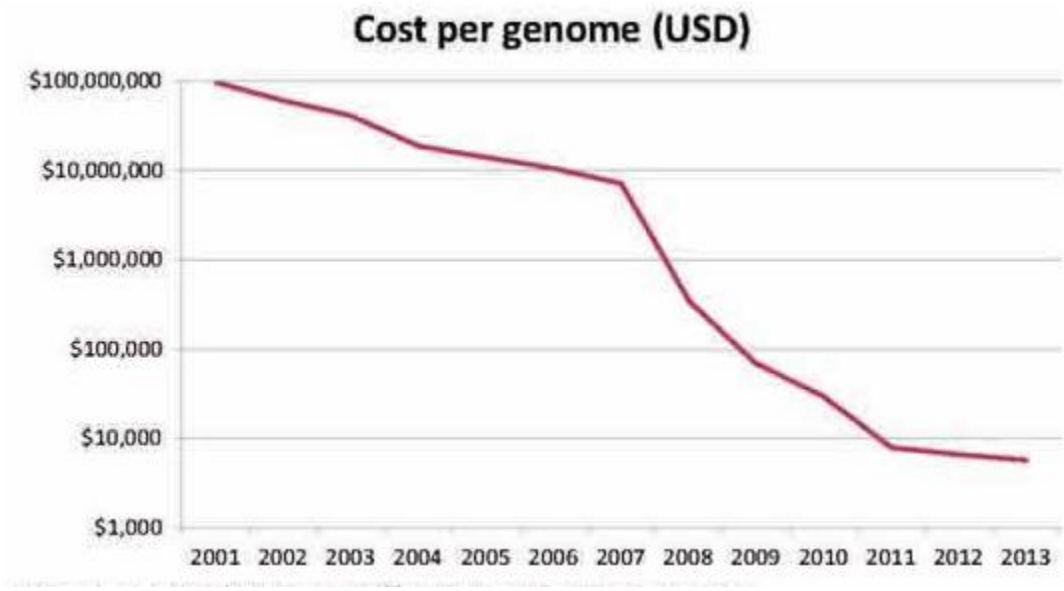


Figure 5 Declining cost of human genome sequencing [3].

[3] Source: National human genome research institute.

We are going to show the characteristics and laws of parallelism, architectures of parallel computers, hardware and software solutions to achieve parallelism and the problems with communication systems.

2.2.1. Characteristics

In this section, we identify the characteristics used to describe parallel systems. These characteristics can be classified in four groups that refer to: (i) the visibility of parallel resources or the concept of parallelism within the system, (ii) the variety of processing elements used to compose the parallel system, (iii) the granularity of parallelism provided by the system, and finally, (iv) who has the control of parallelism.

First group. Macroscopic vs. Microscopic.

This group deals with how parallelism is conceived.

The term microscopic relates to describing parallelism that is not necessarily visible. To be more specific:

- Microscopic parallelism refers to the use of parallel hardware within a specific component. Some examples of microscopic parallelism are:
    - Parallel operations in an ALU.
    - Parallel access to general-purpose registers.
    - Parallel data transfer to/from physical memory.
    - Parallel transfer across an I/O interconnection system.

On the other hand, the term macroscopic relates to the concept of parallelism within the system as the leitmotiv for the system design. Some examples of macroscopic parallelism are:

- Duel-core (multi-core) Intel processors

- NVIVDIA Graphic devices

Second group. Symmetric vs. Asymmetric.

This group deals with the main components with which the parallel system is built. It can be also referred as homogeneous and heterogeneous.

The term symmetric (or homogeneous) means that the main processors of the system are all the same. Some examples of symmetric parallel systems are:

- Duel-core (multi-core) Intel processors.
- NVIVDIA Graphic devices.

On the other hand, the term asymmetric (or heterogeneous) means that there is (some) variety within the processors found in the system.

Third group. Fine-grain vs. Coarse-grain.

This group deals with at which level parallelism is achieved. The term fine-grain (or small-grain) means that parallelism is achieved at (single) data or (single) instruction level (where non-dependent data can be processed in parallel). Some examples of fine-grain level are:

- Matrix multiplication using row and column method.
- Computing single pixels.

On the other hand, in the coarse-grain (or large-grain) level parallelism is achieved at entire routines levels (programs or large portions of data). Some examples of coarse-grain level are:

- Computing FFTs
- Computing frames

Fourth group. Implicit vs. Explicit.

This group deals with who is in charge to control parallelism. In implicit parallelism the control of parallelism is assumed by the programmer.

On the other hand, explicit parallelism (or transparent) does not require a programmer to take the control of parallelism.

2.2.2. Laws

So far, to parallelize an application appears as the magical solution to avoid all the problems related with computational time spent to solve a problem, energy consumption expended by the parallel system to solve a problem, and so on.

However, to achieve a useful parallelism is tricky. According to John Harper[4]:

''Building multiprocessor systems that scale while correctly synchronizing the use of shared resources is very tricky, whence the principle: with careful design and attention to detail, an N-processor system can be made to perform nearly as well as a single-processor system. (Not nearly N times better, nearly as good in total performance as you were getting from a single processor). You have to be very good — and have the right problem with the right decomposability — to do better than this.''

In a more formal way, Gene Amdahl [1] expresses his point of view in 1967 using the following formula:

$$SpeedUp = \frac{(St + Pt)}{(St + \frac{Pt}{N})}$$

where St means the time spent in serial computation by a serial processor, Pt the time spent by a serial processor in the parts of the program that can be processed in

[4] http://www.john-a-harper.com/john-harper-resume-20110901.pdf

parallel, so therefore St + Pt is the total time used computing, and N is the number of processors.

In other words, the Amdahl's law says that the maximum speed up that can be achieved by any process is limited to the weakest link part of this process. Usually, any process can be divided in two parts:

- Serial sections, always present.
- Parallel code, not always present.

Therefore, the speedup of a process using multiple processors in parallel computing is limited by the time needed for serial sections (weakest part in a parallel process) of the process. For example, if we pay attention to Figure 6 we can see that a certain application can be divided in 5 computing modules and each of them consumes 100 unit times to be executed.

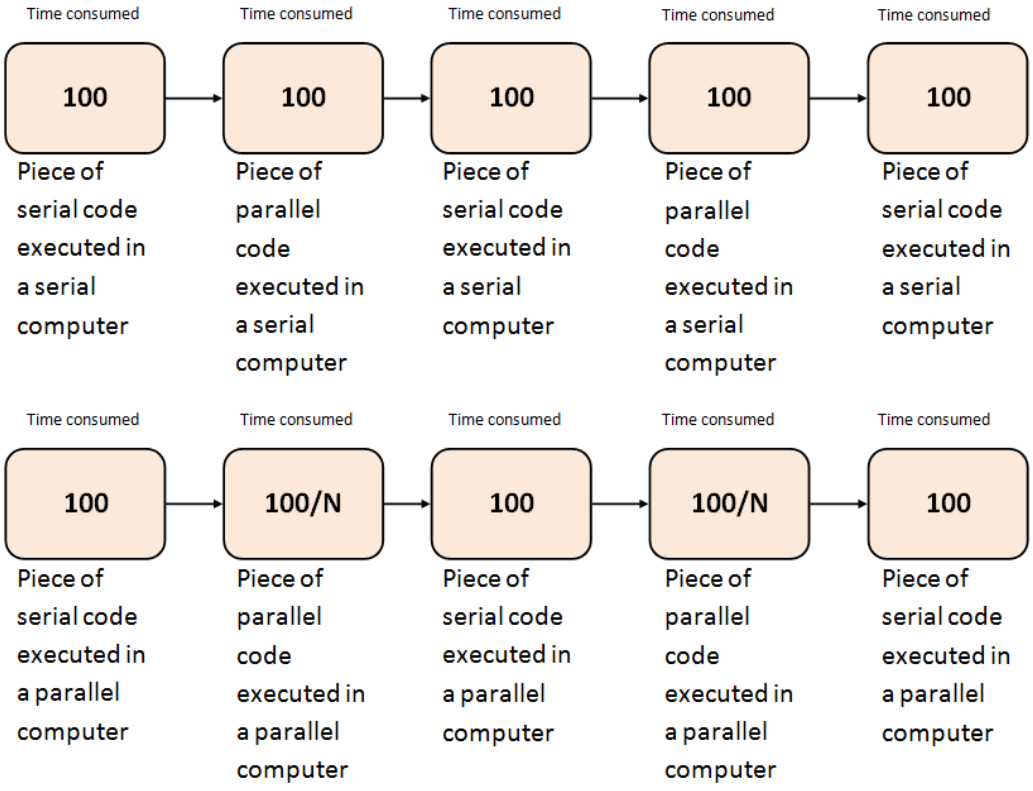


Figure 6 Speedup on Amdah's law.

When executed in a serial computer, the total time spent by the application is 500 time units. However, this application has two modules that can be executed in parallel Therefore, the time required by these two modules when executed in a parallel system is 100 time units divided by the number of processors used to compute those modules. At the extreme, using an enormous amount of processors, the time required by these two modules can be considered 0 (since 100 divided by N will be some value close to 0). In this case, the total time spent by the application will be 300 time unit versus 500 time units used by the serial system. That means a speedup of 1,66 using so many processors, just saved the 40% of the time.

In 1988, at Sandia National Laboratories, John L. Gustafson et al. [21] presented their research involving massively-parallel processing to fight scepticism regarding the viability of massive parallel systems due to the Amdahl's law. Therefore, Gustafson stated the law known as Gustafson's Law, which states that increasing the number of processors gives linear speed up. More processors allow larger dataset size.

For example, using the same previous case, where a certain application, in a serial computer, uses 100 time units to compute a certain amount of data in serial and parallel pieces of code, Figure 7.

When the application is performed in a parallel computer, in the pieces of code that can be parallel, it can be run using more data information and therefore, if in the parallel part 10 threads of dataset are used, the speed up achieved is over 2,17 (Speedup = (500*10)/(100*3 + 100*10*2).

So the difference with Amdahl's law is that Gustafson proposed fixed time and increased work load. Therefore, serial parts of the program have a diminishing effect in reducing the overall speedup in a parallel environment.

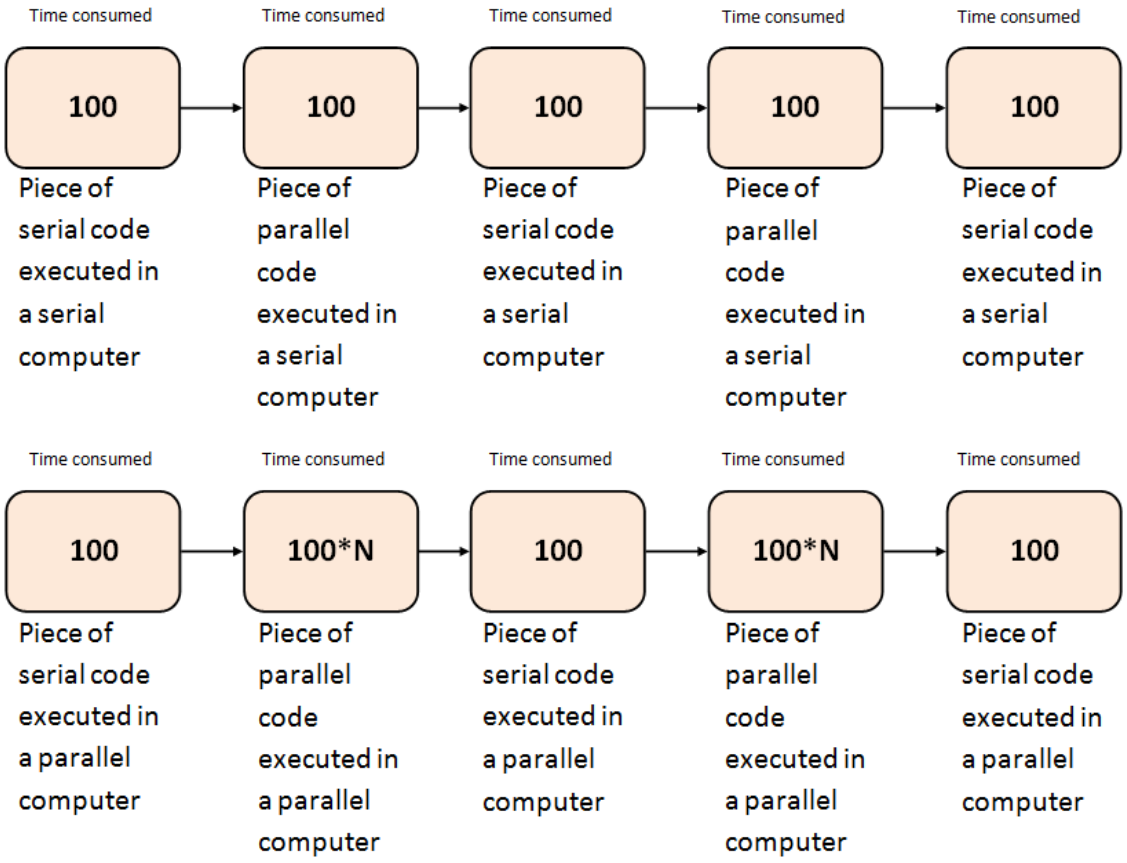


Figure 7 Speedup on Gustafson's law.

His tests revealed that, as processors grow, the problem size is scaled and this scaling results in a substantial increase in the parallel parts of the program as compared to the serial parts. In other words, Amdahl fixed work and Gustafson fixed work per processor.

2.2.3. Hardware Solutions

How parallelism is achieved or supported in hardware? In previous sections we have described the characteristics and laws of parallelism, how we can distinguish and, classify parallelism, but the question now is: how is all of this supported in hardware? What is beneath the surface?

While the clock frequency of the CPU was rising, the instruction throughput per second was also incising. So, the main technique to increase performance was clock

frequency scaling. However, there are other techniques to increase the performance of the processor, for instance parallelism that does not increase power consumption in the same way that clock scaling.

When talking about single processors, parallelism is achieved using pipeline. The pipeline technique allows the execution of several instructions simultaneously. This is achieved by dividing instructions in steps or stages that are processed sequentially one to next in a pipe-like shape. The first step enters first, following by the second stage of the instruction, and leaves first, followed by the second stage, and so on.

This technique does not execute a single instruction in parallel; however it allows to execute several instructions at the same time and to increase the instruction throughput Figure 8. Therefore, the instruction throughput is defined as the frequency of the instructions leaving the pipeline.

					Current Instruction				
Cycle 1	IF	ID	EX	MA	WB				
Cycle 2		IF	ID	EX	MA	WB			
Cycle 3			IF	ID	EX	MA	WB		
Cycle 4				IF	ID	EX	MA	WB	
Cycle 5					IF	ID	EX	MA	WB

Figure 8 Five stages deep pipeline example [93].

The pipeline technique imposes some restrictions to be used properly. Since any stage can be processed at the same time with any other stage, the time required to process any stage is determined by the time required to process the slowest pipe stage. When the processor has a balanced pipeline stages then, the time per instruction on the processor follows the formula: $TPI = (Ti_{NP})/(N)$

Where TPI is the time per instruction, Ti_{NP} Time per instruction on the non-pipelined machine, and N is the Number of pipe stages.

The speedup that can be obtained from the use of the pipeline technique is equal to the number of pipe stages, but only if the stages are perfectly balanced. The balancing of the pipe is a duty of the pipe designer. Usually, the stages are not perfectly balanced, since the pipeline introduces some overhead.

Pipelining is the parallelism technique used for mono-processor. But there are other ways to support parallelism in hardware.

The most common way to implement a parallel system is to create a multi-processor system. Since clock frequency scaling is not anymore a viable solution (due to the fact that we are reaching the physical limits), the focus has been moved to scale the number of cores available per processor.

There are three main types of multi-processor systems based on their memory architecture: (i) shared-memory, (ii) distributed-memory, and (iii) hybrid shared-distributed-memory. We define multi-core systems when the number of processor cores is less than 16. When there are more than 16 then we used the term many-core system. Some examples of many-core systems are the NVIDIA GPUs, or the research platform from ST Microelectronics STHORM. Later on this document, we will go deep on many-core systems details.

The last way of parallelism in hardware is the multi-computers systems (or clusters). Cluster systems have long been used in scientific computing, where large problem sizes could not be solved on a single processor. In the 70s, the first supercomputer was introduced by Seymour Cray. The first Cray-1 supercomputer system was installed at Los Alamos National Lab for 8.8 million dollars, and achieved 160 megaflops and had 8 mega byte main memory [22]. In 2011, Cray installed the Jaguar supercomputer

supported 200 cabinets of Cray XT5 blades, and each cabinet can be hold 24 Cray XK6 blades. That is 307200 CPU cores with 16 petaflops. Multi-computer systems can be created by interconnecting through a large network (LAN, internet) a large number of full computers, such a PCs, laptops, workstations, and so on. This can be called as grid computer. Examples of multi-computer systems are the Condor platform from UAB for instance.

Supercomputers have been essential in cryptography tasks. Figure 9 shows the breakdown of uses of supercomputers systems.

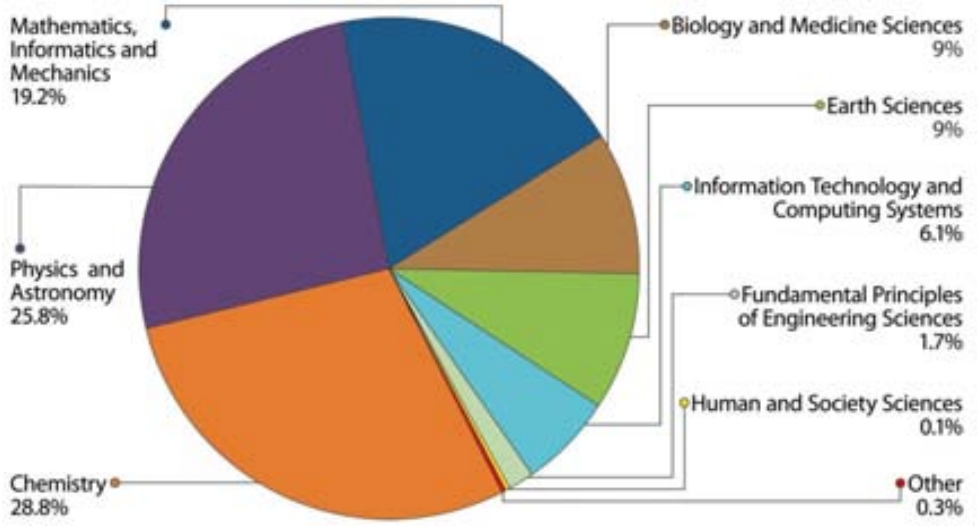


Figure 9 Breakdown of uses of supercomputer systems[5].

2.2.4. Theory of Architectures

In this section, we enumerate the different options of available architectures, and following the classifications on the theory, we select the suitable one for our MPSoCs.

Following Flynn's taxonomy [23][24], computer systems may be classified according to global control, and data and control flows. In practice, Flynn's taxonomy distinguishes between four categories:

[5] Source: Moscow University Supercomputing Center. http://hpc.msu.ru/?q=node/77

- Single-Instruction Single-Data (SISD). When a serial computer does not exploits any kind of parallelism. A single control unit fetches a single instruction from memory. The most evident examples of this category are the classical single-processor machines used as a PC like the desktop computers. It must be said that nowadays PC systems have multiple processors.
- Multiple-Instruction Single-Data (MISD). When a computer system exploits a single data flow in several instruction flows. This category is used in fault tolerant systems.
- Single-Instruction Multiple-Data (SIMD). When a computer exploits several data flows in a single instruction flow, to process parallel operations. GPUs are an evident example of this category.
- Multiple-Instruction Multiple-Data (MIMD). When a computer exploits several data flows in several instruction flows. Multiple processors executing in parallel different instructions using different data.

We select MIMD because it has multiple processing elements with separate data and instruction access to data and program data memory (shared or distributed). In MIMD each processing element can execute its own program; therefore, it is a more flexible architecture than SIMD, with only one program memory, and MISD, with one common access to a single global data memory. SISD is not even considered here since there is not a multi-processor.

Nearly all general-purpose parallel computers follow the MIMD model. However, according to the memory organization and to the different point of view from which it can be seen (physical or the programmer's), MIMD may be further classified. From the programmer's point of view, we must distinguish two cases: distributed address space or shared address space. It needs to be clarified that this classification does not match necessarily physical classification.

From the physical point of view, a memory may be classified as a shared memory or distributed memory, even though there are hybrid organizations that can provide, for instance, a virtual shared memory on a physical distributed memory.

In shared memory organization, processing elements can operate independently while sharing the same memory resources, which becomes the source of several problems. Shared memory architectures may be mainly divided into two classes based upon the memory access time: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA).

In UMA architectures, using an indirect network to access the external memory imposes a uniform latency to access the memory, which could be a counterproductive requirement in a large MPSoC since some processors will be much closer to external memory than others will and, consequently, performance will drop.

NUMA architectures mitigate this scalability problem presented in UMA architectures. In NUMA (ccNUMA when cache coherence is used) architectures, not all processors have equal access time to all memories. However, ccNUMA ultimately suffers the bottlenecks imposed by cache coherence protocols as indicated by Heinrich et al in [25].

Despite the fact that data sharing between processor elements is faster in shared memory architectures than in distributed memory, the lack of scalability between memory and processor elements is the primary disadvantage of such architectures.

For this reason, we choose distributed memory architectures for our system. These architectures could be a viable way to overcome the limits that cache coherency imposes in terms of performance and in additional hardware support, becoming scalable architectures [26].

In distributed architectures, each core is a complete computer system, where no processor is allowed to access the memory module of another processor; hence, these systems can be called No Remote Memory Access (NORMA) architectures.

However, since the chip pinout is limited and external memory is mandatory for MPSoCs, not every core will be able to access the external memory.

Consequently, homogeneous NORMA, as depicted in Figure 10 (left), will be only feasible for a very limited set of applications with low memory demand.

Two particular architectures are promising alternatives: COMA (Cache Only Memory Access) and heterogeneous NORMA. In COMA architectures, Figure 10 (right), processors have only cache and the rest of the memory is accessed through special I/O hardware. It might seem strange that we consider COMA to be a distributed memory architecture, but the reason is that interconnect is not necessary tailored to memory transactions. The main problem with this architecture is the need for additional hardware to control the input and output to the off-chip memory.

In heterogeneous NORMA, Figure 10 (middle), only a few processors would access the external memory. These processors have some cache memories and control the other processors of the system, which would access only their local memory. Therefore, distributed architectures solve the problems of scalability existing in shared memory architectures [25][26].

All this considered, for our MPSoC system we select the distributed heterogeneous NORMA architecture, which provides us flexibility – since each core can execute its own program flow– scalability, and does not require any special hardware, as in COMA architectures.

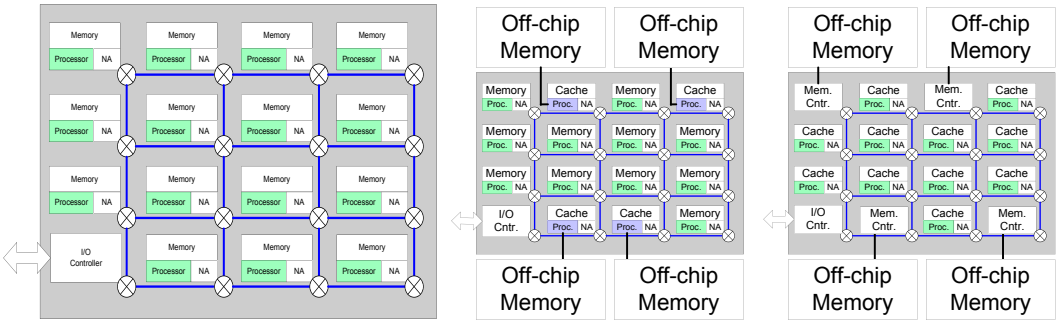


Figure 10 Left) Homogeneous NORMA. **Middle)** Heterogeneous NORMA. **Right)** COMA.

2.2.5. Software Solutions / Levels of Parallelism

From the software point of view, the parallelism can be classified on two different levels of parallelism:

- ILP. Instruction-level parallelism. When several instructions from the same instruction stream are executed in parallel.
- TLP. Thread level parallelism. When several threads from the same application are executed in parallel.

Multithreading attempts to push the utilization of functional units further by sharing functional units between more than one thread. Each thread has its own copy of the PC, register files, GPR, and so on

Thread switching time needs to be optimized. Virtual memory supports the sharing of memory resources. Two basic methodologies,

- Fine-grained multithreading:
    - Threads interleaved on an instruction by instruction basis
    - Stalled threads are ignored.
    - Advantage,
        - Smoothes out any stall cycles.
    - Disadvantage,

        - Increases the latency of any single thread.

- Course-grained multithreading:
    - Only switches between thread execution on a 'costly' stall (e.g. a cache miss)
    - Advantage,
        - Prioritizes the throughput of a specific thread.
        - Simpler to control!
    - Disadvantage,
        - Each thread switch encounters a startup overhead (empty pipe).
        - Overall throughput and individual thread latency both poor
    - Not used in commercial CPUs

2.3. Theory of interconnection architectures

All this considered, for our MPSoC system we select the distributed heterogeneous NORMA architecture, which provides us flexibility – since each core can execute its own program flow – scalability, and does not require any special hardware, as in COMA architectures.

Future embedded System-on-Chip (SoC) will probably be made up of tens or hundreds of heterogeneous Intellectual Properties (IP) cores, which will execute one parallel application or even several applications running in parallel. These systems could be possible due to the constant evolution in technology that follows the Moore's law, which will lead us to integrate more transistors on a single die, or the same number of transistors in a smaller die.

For such SoCs, Network-on-Chip (NoC) architectures are the solution for the scalability problem. Traditional on-bus communication-based solutions, **Figure 11**a,

pose serious problems related to the integration of several IP cores. As the number of components connected to the bus raises, the bus system will produce a performance bottleneck problem [16] appears in the bus system..

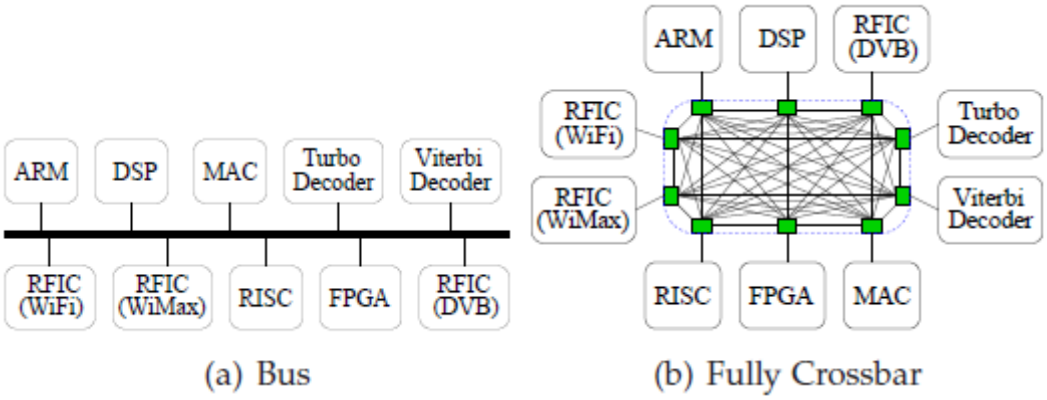


Figure 11 Interconnection systems a) Bus-based system approach b) Crossbar-based system approach.

An alternative for on-bus communication solution is to substitute the bus connection by a fully crossbar system, Figure 11b, but as the number of participating components rises, the complexity of the wires could be dominant over the logical parts.

Finally, NoC-based interconnection system was presented as the solution to these problems. The NoC [16][17][18][19][20] entails a unified solution to the On-Chip communication and the possibility to do scalable systems at supportable levels of power consumption. In embedded MPSoC systems, NoCs can provide a flexible communication infrastructure, in which several components such as microprocessor cores, MCU, DSP, GPU, memories, and other IP components can be interconnected. NoCs have been extensively discussed in several regular publications and special issues, from journals to conferences and workshops, and also, the NoC topic has inspired symposiums like ACM/IEEE International Symposium on Network-on-Chip.

Figure 12 shows the evolution of the interest the NoC topic has raised in the last decade in terms of hits when “network-on-chip” is searched in the IEEE Xplore digital

library [27]. 0 also shows the evolution of a nother two topi cs combined with NoCs: multiprocessor systems based on ne twork-on-chip int erconnection s ystem publi cations have be en incr easing it s popularity year by year, whereas the other t opic, parallel programming model for such systems, seems not to be increasing its popularity among the academia, despite the interests shown by the industry in such topic.

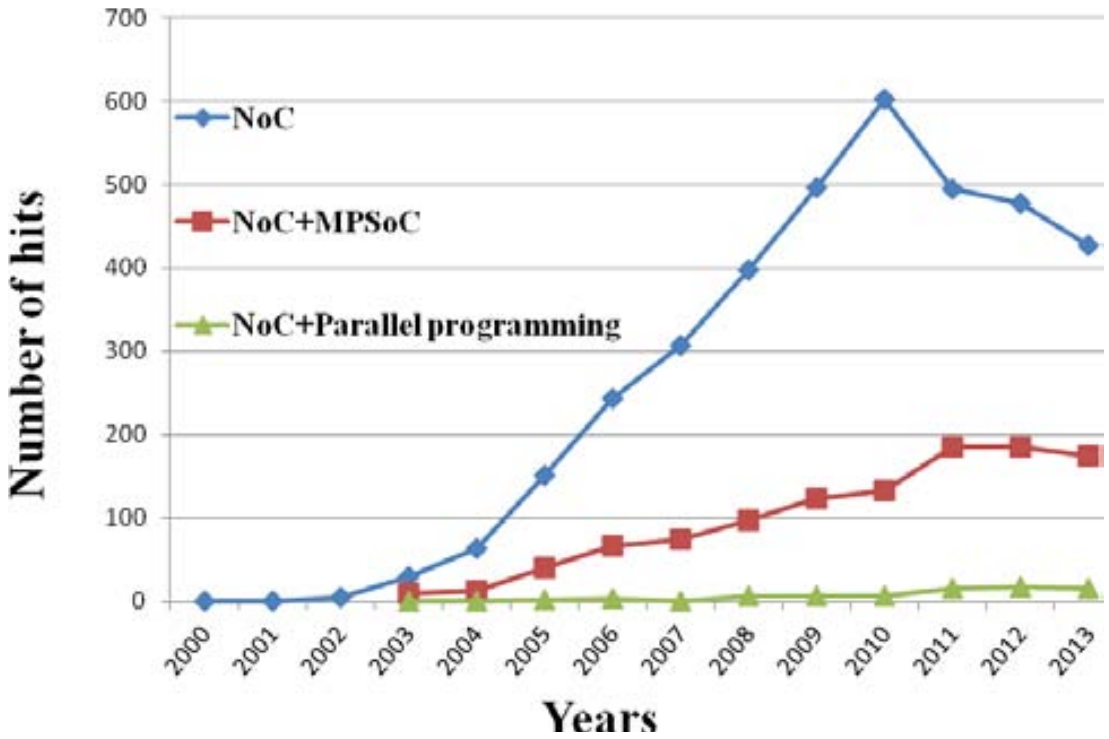


Figure 12 IEEE Xplorer hits for different "network-on-chip" searches

2.3.1. NoC Components

Basically there are just three main components on a NoC-based system (Figure 13):

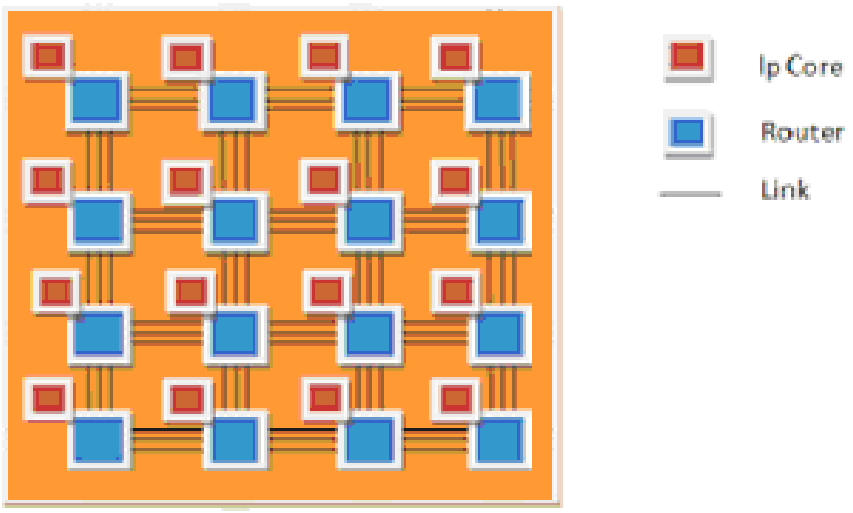


Figure 13 NoC representation.

- Network Interface Controller (NIC). The NIC implements the interface between each IP node and the communication infrastructure. The architecture of the NIC component can be divided into two modules. The first one is focused on the interaction with the computation or memory node bus, and the other one is focused on the interaction with the rest of the NoC. This component is called in many different ways in NoC literature: NI for Network Interface, NA for Network Adapter, or RNI for Resource Network Interface are some examples.

  Router (Figure 14). Also called switch. These components are in charge of forwarding data to the next tail. On the routers we can find the routing protocol, buffer capabilities and the switching method. In general, the router component is composed of the following elements: (i) Arbiter, whose main task is to grant channels (selecting an input port and an output port) and route packets; (ii) Crossbar of n input x n output ports that direct the input packet to the corresponding output port; and if that is the case, as it is in the packet switching protocols, (iii) a buffer or queue, which is used to buffer incoming and outgoing data in the router.

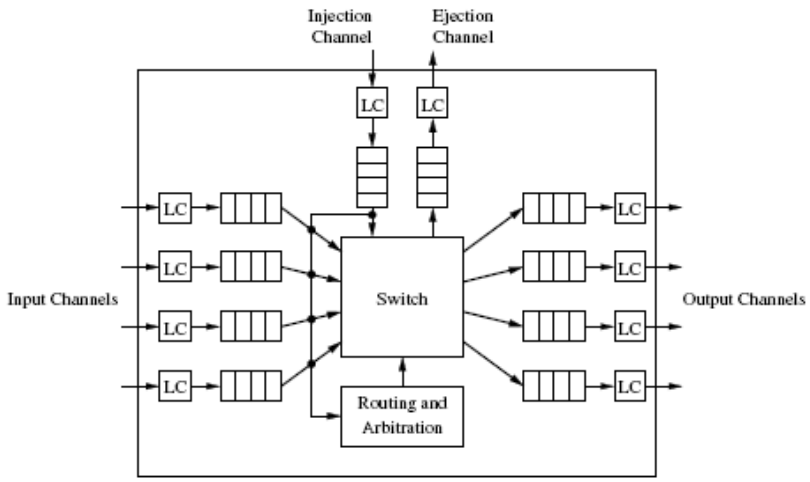


Figure 14 Generic Router block diagram from DUATO[91] (LC = Link Controller)

- Links. Links are the specific connections that provide communication between components.

2.3.2. Switching Method

Traditional multiprocessor networks techniques have been adapted to on-NoC-based multiprocessor systems. There are basically two switching methods: Circuit-Switching and Packet-Switching.

In Circuit-Switching, a path from source to destination is reserved before the information is emitted through the NoC components. All data are sent following the reserved path which is released after the transfer has been completed.

In Packet-Switching, there is not any reserved path from source to destination; instead, data are forwarded hop by hop using the information contained in the packet. Thus, in each router the packets are buffered before being forwarded to the next router.

In Packet-Switching we can distinguish three choices according to how packets are stored and forwarded to routers: (i) Store-and-Forward, (ii) Virtual Cut-Through and (iii) Wormhole.

In Store-and-Forward protocol, the packet is stored completely before forwarded to the next hop. Thus, if the router in the path does not have sufficient buffer space, the packet is stalled. This method requires buffering capacity for at least one full packet.

In Virtual Cut-Through protocol, the packet is forwarded to the next hop once it is guaranteed that the full packet can be stored. The main difference with Store-and-Forward is that there is no need to wait for the storage of the complete packet before forwarding it to the next hop. However, this method also requires buffering capacity for at least one full packet.

In Wormhole protocol, each packet is further divided into small units called flits. There are three different types of flits; header, body and tail. The header flit reserves a path between hops and establishes a channel where one or many body flits – which contain the packet information – follow, and finally the tail flit releases the reserved path. The major advantage of Wormhole method is that there is no need of buffering capacity for a complete packet.

2.3.3. Topology

Since NoC interconnection systems are replacing traditional bus interconnection systems, many topologies have been proposed, most of them adapted to the constraints of the embedded world from parallel multiprocessors systems. Topologies can be classified following geometric criteria as:

- Regular topologies. Mesh-like, fat-tree, ring, torus or star, are examples of regular topologies. (Figure 15)
- Irregular topologies. Custom or application-oriented designs as shown in 0 .

Topologies can also be divided into networks where all nodes are attached to a core and networks where they are not:

- Direct topologies. In direct topologies all nodes are attached to a computational or memory core. Mesh, torus or rings are examples of direct topologies. (Figure 15)
- Indirect topologies. In indirect topologies not all nodes are attached to a core. Trees or star topologies are examples of indirect networks. Figure 15

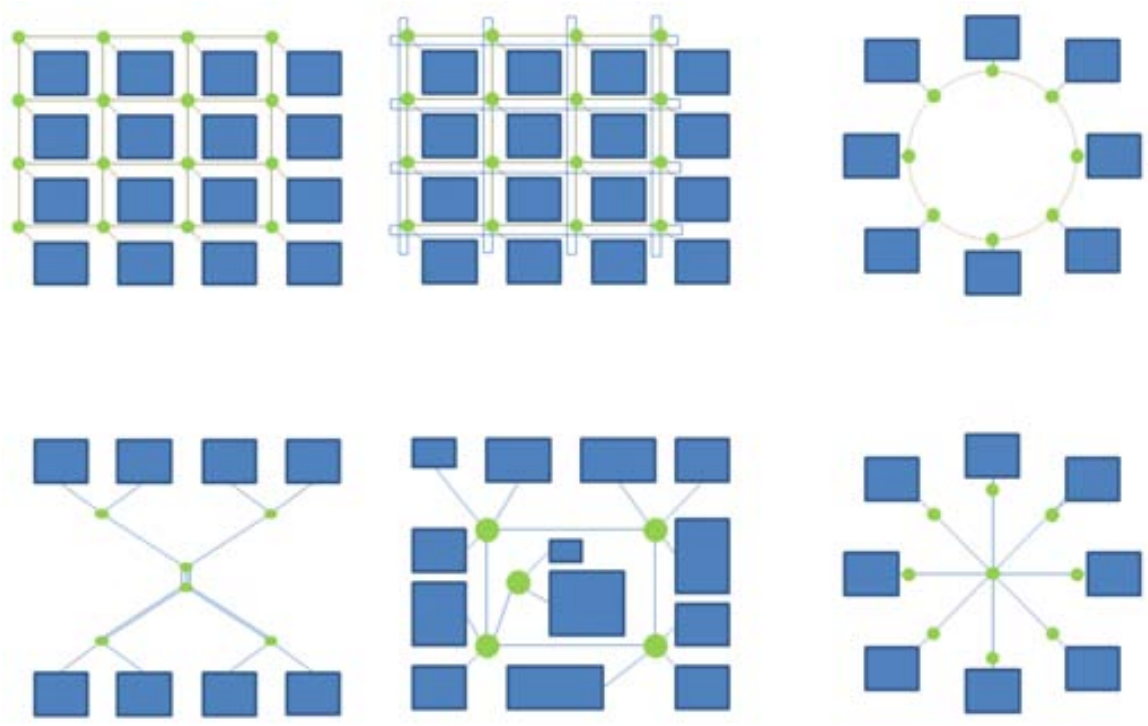


Figure 15 Some basic network topologies. a) Mesh (up-left), b) Torus (up-middle), c) Ring (up-right), d) Fat-tree (down-left), e) Custom (down-middle), and f) Star (down-right). All links are bidirectional.

2.4. Parallel Programming Models

As it has been remarked previously, the main reason to adopt the NoC paradigm is due to the scalability issue.

However, there are many other variables to take into account, particularly adopting NoC architecture. Having a scalable communication system is not enough to achieve fully scalability, since it is mandatory for the programmer to have enough tools to design applications that will run efficiently on MPSoC systems.

The programming model is the necessary way that permits programmers to abstract the logic of applications and translate it to the hardware platform system. Programming models are the bridge that must save the gap between hardware and software trying to raise productivity and efficiency. Therefore, if the programming model is developed from the hardware point of view (that is bottom-up) then programming the system could be a tough task decreasing productivity. If the programming model is developed the other way around with a top-down approach, then the efficiency could be affected due to the difficulties that could appear when mapping the application in the hardware system.

A programming model must provide scalability – that is, to ensure the performance increase of the system when increasing the number of hardware resources available in the system.

2.4.1. Traditional Parallel Programming Models

Typically, there are two traditional parallel programming models:

- Shared-memory model: when communication occurs implicitly through a global address space accessible for all processors. This model implies to ensure data coherence and synchronization. Systems based on this model usually have a shared memory architecture, which can suffer a performance bottleneck due to the memory hierarchy.
- Message passing: when communication occurs between a sender and a receiver. Message passing model implies a set of processors with no shared address and also implies collaboration between sender and receiver. The most common primitives used for communication in this model are send and receive, and always a send operation must match a receive operation. This model could overcome the non-

determinism and the scalability limits that cache coherence protocols introduce in shared memory architectures. The main drawbacks of message passing model are that the programmer must explicitly implement the parallelism and data distribution dealing with data dependencies and inter-process communication and synchronization.

Other parallel programming models are:

- Data parallel model: when data partitioning determines parallelism and several processors perform the same operation concurrently.
- Thread-based model: when a process have multiple threads running concurrently.

2.4.2. Programming Models for NoC-Based Systems

A large number of MPSoC-specific programming models have been defined in the last years based on shared memory or message passing models. Examples of programming models will include OpenMP [28] for shared memory architectures, or MPI [29] for message passing. Below, we will describe some of these existing parallel programming models implementation.

2.4.3. Shared-Memory Programming Models for NoC-Based Systems

OpenMP: for open multi-processing. OpenMP is an API for shared-memory multiprocessing in C, C++ and Fortran. In OpenMP all threads can access the shared data, but private data can be accessed only by the thread that owns such data. OpenMP expresses parallelism using a set of compiler directives called #pragma. OpenMP is supported on Cell processor for example.

CUDA[30]: for compute Unified Device Architecture. CUDA, provided by Nvidia, is an example of programming model used in industry. CUDA is a software platform for parallel computing in C, C++ and Fortran on Nvidia GPUs (graphic processing unit). CUDA requires the programmer to write special code for performing parallel processing.

OpenCL[31]: for open computing language. OpenCL is an open standard for parallel applications over multi-core platforms with CPUs and GPUs. OpenCL is developed by Khronos group [32], which is formed by several industrial partners as, for instance, IBM, ARM, AMD, or Intel. OpenCL is based on the model of one host plus one or more computing devices, which are a collection of one or more CPUs or GPUs. In OpenCL execution model the code for an OpenCL device is written in C and it is called kernel, and a collection of kernels and other functions is a program. OpenCL provides APIs for writing kernel in C, and APIs are used to define and control the platforms that are the hardware abstraction of diverse computational resources. In OpenCL the memory management is explicit and it is responsibility of the programmer to move data from host to the computer device global and local memory, and back.

2.4.4. Message Passing Programming Models for NoC-Based Systems

MCAPI [33]: for multi-core communications API is a research work from Multicore Association that defines a set of lightweight multi-core communication API for closely distributed embedded systems (multiple cores on a chip). MCAPI provides three modes of communication: messages, connected-channels packets and connected-channels scalars. MCAPI is independent from language, processor and operating system.

MPI : for message passing interface. MPI has been recently adopted as a standard “de facto”. It basically specifies a set of point-to-point and collective communication primitives, and creation and management of process primitives. MPI is language-independent, and recently a large number of traditional message passing interface programming models are being proposed for MPSoCs, which are discussed in chapter 4.

2.4.5. Other Parallel Programming Models Implementations

TBB [35][36] : for Intel Threading Building Blocks, it is a commercially supported open-source C++ template library for shared-memory programming model.

X10 [37] is a class-based object-oriented programming language from IBM.

Pthreads: for POSIX Threads, it is a POSIX standard for threads.

StreamIT [38]: from MIT, it is a programming language specifically designed for streaming systems.

Cilk/Cilk++ [38][39] is a C-based runtime system for shared-memory parallel programming developed by MIT.

Chapel [40]: from Cray, it is a parallel programming language developed as an open source with contributions from academia and industry.

Axum [41]: from Microsoft, it is a programming model based on .Net.

3. Simulation Environment and MPSoC system implementation

In order to validate the work done and the results obtained, it is necessary to know about the way how those resultants are found.

In this chapter, some short details about the simulation tool used to obtain the results that will be analysed in further chapters are given: NoCMaker [42]. Additionally, in this chapter we present the NoC-based MPSoC system that provides a basis for the off-loading work shown in this dissertation.

3.1. NoCMaker

NoCMaker is an open source architectural tool to explore the huge network-on-chip design space. It is based in JHDL [94] and has been developed within the ITEA-Project PARMA [43] at CEPHIS (UAB) [44]. NoCMaker allow designers to create cycle accurate designs of different NoC systems, to validate and synthesize them. Since there is a huge number of variables interacting (such as traffic distribution, network topology, switching scheme, flow control, routing algorithm, channel properties, FIFO depth, packet/flit size, and so on), the NoC performance cannot be determined a priory. These variables compose a multidimensional design space that can be explored with NoCMaker.

JHDL (Java Hardware Description Language) is a hardware description language (HDL) based on Java, that is attached to a set of FPGA tools that allow the user to design the structure and layout of a circuit, to debug the circuit in simulation, and to generate an Electronic Design Interchange Format (EDIF) netlists and interfaces for

synthesis. JHDL allows NoCMaker to estimate the number of resources (Les for Altera FPGAs) used by the NoC. JHDL is a hardware description language (HDL) based on Java. There are some good reasons to use JHDL to describe NoCs:

1. Circuits can dynamically change their interfaces. This eases the design of some system elements like switches.
2. Block construction can be parameterised by complex arguments, simplifying the context information needed by the module creators and enabling the design of powerful traffic generation modules.
3. Custom state viewers can be developed to provide a much richer interpretation of system state than waveforms.
4. Simulation is interactive.
5. Java is cross-platform.

It is defined a Java class to represent NoC design space points (NDSP), which can be retrieved from XML, and NoCMaker use objects of this class to create and handle their respective NoC instantiations.

Once NoCMaker is running, a NDSP characterized in XML can be loaded, or a wizard GUI opened to define step by step all the parameters of a new DSP. Following these lines, Figure 16 shows an example of a XML code to create a 4x4 Wormhole switching NOC with a four-phase handshake distributed XY routing algorithm.

```
<org.cephis.nocmaker.model.NoCDesignSpacePoint>
  <outputChannelArbitration>DEMAND_SLOTTED_RR</outputChannelArbitration>
  <topology>MESH</topology>
  <queueingType>INPUT</queueingType>
  <removeInputLocalQueue>false</removeInputLocalQueue>
  <removeOutputLocalQueue>false</removeOutputLocalQueue>
  <queueLength>8</queueLength>
  <switchingMode>WORMHOLE_SWITCHING</switchingMode>
  <routingDecision>DISTRIBUTED</routingDecision>
  <routingAlgorithm>XY</routingAlgorithm>
  <flowControlMethod>FOUR_PHASE_HANDSHAKE</flowControlMethod>
  <clocking>GLOBAL_CLOCK</clocking>
  <trafficPattern>BIT_REVERSAL</trafficPattern>
  <busType>ABSTRACT</busType>
  <processorType>ABSTRACT_32_BITS</processorType>
  <packetSourceAddressBits>8</packetSourceAddressBits>
```

```
  <packetDestinationAddressBits>8</packetDestinationAddressBits>
  <packetPayloadSizeBits>0</packetPayloadSizeBits>
  <packetMinPayloadBits>224</packetMinPayloadBits>
  <packetMaxPayloadBits>224</packetMaxPayloadBits>
  <channelWidth>32</channelWidth>
  <injectionRatio>0.010</injectionRatio>
  <injectedBytes>2800</injectedBytes>
  <dyadRouting>false</dyadRouting>
  <meshWidth>4</meshWidth>
  <meshHeight>4</meshHeight>
</org.cephis.nocmaker.model.NoCDesignSpacePoint
```

Figure 16 XML example of a NDSP

NoCMaker application produces a fully functional visual JHDL model useful for verifying and viewing the performance and behaviour of the network on-chip.

The graphic feature provided by the tool includes: a schematic view, which allows the designer to navigate through the hierarchical layers of the NoC at RTL level (Figure 17), a cycle accurate simulation sphere to simulate any created circuit (Figure 18), a waveform view, a time diagram of messages (Figure 19) and a point-to-point traffic analysis (Figure 20).

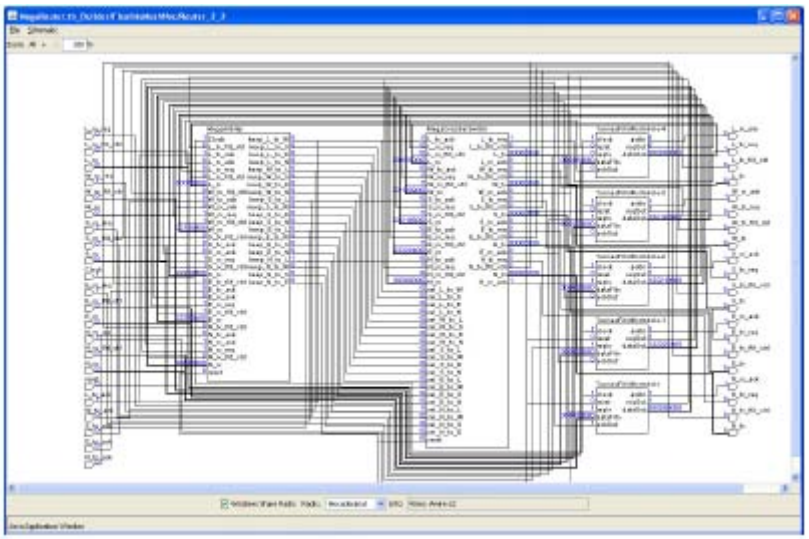


Figure 17 RTL view from JHDL schematic view of a router.

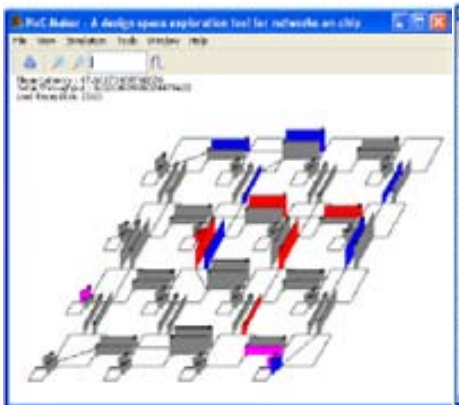


Figure 18 Visualisation of traffic loaded on links during interactive simulation.

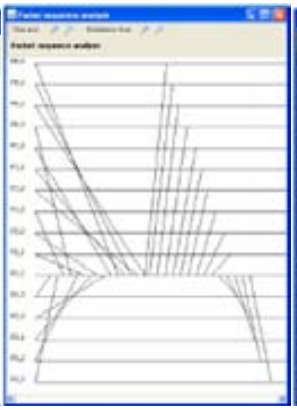


Figure 19 Time diagram of messages.

Figure 20 Point to point traffic analysis example.

NoCMaker tool provides estimations for latency, throughput, and used resources. These parameters can be computed using statistics of all packets that go through the network. The tool allows creating NoC with the following (limited) characteristics:

- Switching methods: Wormhole and Stall-and-go.
- Topology: Mesh, Tree, Ring, Bus.
- Flow control: Stall-and-go, Four phase handshake.
- Bus type: Abstract processor bus, Avalon bus, OPB.
- Routing algorithm: Centralised, Distributed, Adapting, Static.

3.2. MPSoC System

The constant evolution in technology makes possible the design of complex multiple-processor systems that can integrate more and more IP (Intellectual Property) cores on a single SoC (System-on-Chip). The more IP cores connected to a bus, the more complexity in the bus arbitration, the more latency introduced, the more the performance drops. Finally, the evolution of the On-Bus-based communication to On-Network-based communication architecture [45] has been presented as the solution to these problems. Following this approach, commercial ASIC multiprocessors like Intel

SCC (Intel Single Chip Cloud), Tilera TILE or NVIDIA Fermi have become an important part of the state-of-the-art, making the research focus of industry and academia to expand into other issues like programming models and languages. On the other hand, and like most types of integrated circuits, FPGAs (Field Programmable Gate Array) are profoundly affected by technological advances that maintain the rate of increase in transistor integration density predicted by Gordon Moore in 1965. Over time, increasingly big FPGA devices have been successfully commercialized by manufacturers and exploited by hardware designers. Figure 21 shows the amount of resources, both logic and memory blocks, that a FPGA manufacturer like Altera has been able to fit in its biggest commercial chips over the last decade. Recent devices already offer more than 1 million Logic Elements (LEs) and more than 40 Megabits of embedded memory. Nowadays FPGAs are used to implement highly complex logic circuits for all kinds of applications.

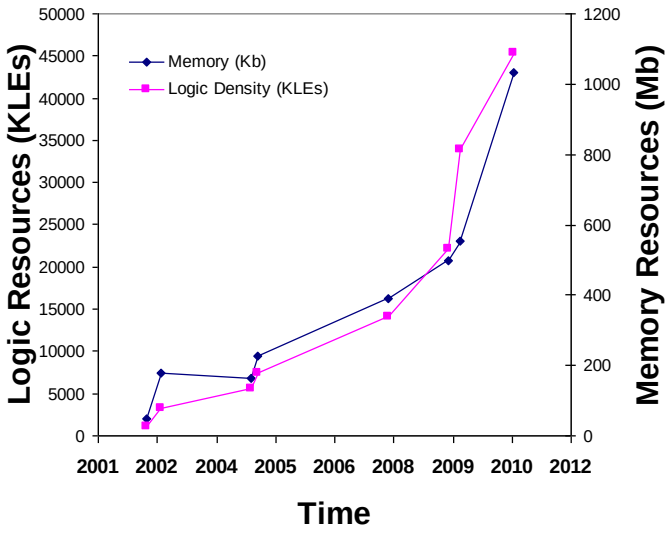


Figure 21 Evolution of the capacity in logic and memory resources in FPGA devices of the Altera Stratix family.

With current resource capacity, it is completely reasonable to anticipate the features and benefits, in terms of processing performance, of building many-{soft}core

multiprocessors on FPGAs, while reducing processor footprints and providing better thermal characteristics. However, this kind of processor has other problems that must be solved in order to realize the massive potential of the MPSoC (Multiprocessor System-on-Chip). These problems affect all levels of the system; specifically, there are problems related to choosing the proper architecture, problems with IP modules involving different hardware solutions, such as different connectivity options for the PE (processor elements) or for helping the debugging process, programming model problems, programming environment/framework problems, and also problems related to performance analysis methods.

In order to answer these problems, we developed a novel many-core hardware/software co-design framework including tools that help (i) to design the system, (ii) to program a parallel application for such system, and (iii) to analyze the performance of such application.

3.2.1. Related work

Some research has been done in proposing reconfigurable MPSoCs. Dorta et al work [46] provides a remarkable overview of FPGA-based MPSoC designs proposed recently. Tseng and Cheng present a small multi-core system containing 4 Altera NIOS II processors in [48]. Although they provide detailed information about the architecture, it is based on the shared bus and shared mutex approach that Altera provides to build multi-core systems; using a shared bus interconnect obviously limits the scalability of the system, and it is the main motivation of NoC (Network-on-Chip) proposals, as explained in numerous pioneering NoC works. Moreover, they achieve a modest speed-up factor using a very simple benchmark. Wang and Hammami propose a scalable architecture in [49], consisting of 24 Xilinx Microblaze processors and an Arteris NoC

interconnect that give access to 4 external DDR2 memory banks. However, this work does not provide any details about the performance and scalability obtained by the system. Tian and Hammami did a similar work creating a 16 Microblaze processor [50] interconnected with two NoCs, one used for synchronization and the other for data transport. However, this work did not propose any programming model for the MPSoC, nor did it make any application test. MPLEM system, proposed by Mplemenos et al in [47], describes an MPSoC system based on Xilinx Microblaze Soft-Core with up to 80 cores. However, this paper fails to give details about the programming model and the performance obtained by real applications. Moreover, it is not clear how the proposed interconnection system – based on multiple segmented buses – can avoid becoming a bottleneck in the application scalability.

More completed approaches that include an MPSoC system plus a programming model proposal comes from:

- Vanderbauwhede et al works [51][52], where an architecture and programming model for dataflow-oriented applications are presented. In this solution, all elements of the MPSoC are configured according to the analysis of DFG (Data Flow Graph) and CFG (Control Flow Graph) of the program that will run on it. That approach reduces the footprint considerably; however, the use of PIM (Processor-in-Memory) processors and point-to-point link connections makes this solution useless for memory-based applications.
- D. Göhringer et al, in RAMPSoC SoC [53], show a runtime adaptive MPSoC with a proprietary subset of standard MPI (Message Passing Interface) library. Even though this work provides some details related to the topology and to the MPI library, it does not present any information about the scalability obtained, neither an application execution example.

OpenMPI and MPICH are two of the most well-known implementations that support around 300 standard MPI primitives. However, their huge size required by the library makes both implementations completely out of the scope for embedded systems. TMD-MPI, SoC-MPI and RAMPSoC-MPI implementations have been presented as lightweight solutions for embedded systems. The drawback here is that all these implementations are proprietary solutions for specific systems, and therefore, non-portable to other platforms, unlike our ocMPI implementation shown in chapter 4, which is open-source and completely portable to other platforms.

Other works present complete simulation environments similar to our approach.

- Yujia Jin et al, in [54], developed an exploration framework to build FPGA multiprocessors for a target application. However, they limit the design to micro-architectures built from a network of processors interconnected using buses or point-to-point links.
- ESPAM [55] tool allows MPSoC generation from high-level descriptions, but it does not use programmable cores, and only crossbar, shared bus or point-to-point interconnection are available.
- D. Atienza et al [56] provided a HW/SW FPGA-based emulation framework that allows system developers to evaluate designs in terms of energy consumption and temperature. It is an outstanding work, but it presents a very specific framework that does not provide facilities for the programmer of NoC-based MPSoC systems.
- MPARM [57] and HeMPS [58] are similar to ours, since they develop a complete simulation platform for MPSoC. Both works are based on SystemC simulation environments that include models for processors, but they do not present any hint for supporting programming models.

- xENOC [59] is also another environment similar to ours, and it includes an Embedded Message Passing Interface that supports parallel task communication; however, our work goes depth and gives facilities to debug and optimize the code for the programmer.
- Recently, Altera has presented, Qsys [60], a new system development tool that includes support for NoCs. Altera Qsys NoC presents a flexible implementation, parameterizable packet format, low-latency interconnect, and separate command and response networks. Qsys NoC interconnect system has been designed specifically for FPGAs, and it is mainly oriented to shared memory architecture, which has a different approach from ours.

In this work, we explain our proposal to achieve a complete MPSoC system. In Figure 22, one can see the whole process that may be defined in three basic steps: hardware development process, software development process, and analysis process.

In the first step, an architectural idea for an MPSoC is chosen, designed and implemented. In Theory of Architectures section, we explained the different options we have for choosing the overall system architecture, and we selected one as a main option for building MPSoCs. Furthermore, in MPSoC System Example section, we show an example of an MPSoC system explaining how this system has been created.

When the MPSoC has a NoC-based interconnection system, as we propose for such kind of system, NoCMaker [42] tool gives an early feedback of the performance of the NoC. In order to obtain this feedback, NoCMaker tests the interconnection system using synthetic traffic [61] and MPI applications. For the designer it is important to have early feedback to avoid the need to go through slow synthesis process in order to validate the system. Once the system is validated, NoCMaker generates synthesizable Verilog code.

For the software development process, the second step, we provide to the MPSoC software developer a complete software stack based on the OSI (Open Systems Interconnection) model that can extract the whole potential of the MPSoC system. More information about this process is given in chapter 4.

The final step of the proposed process is the analysis process. The design and implementation, and the programming of an MPSoC are just a part of the picture. The need of solving possible software bugs or malfunctions, and the need of software optimization are an essential part of the process.

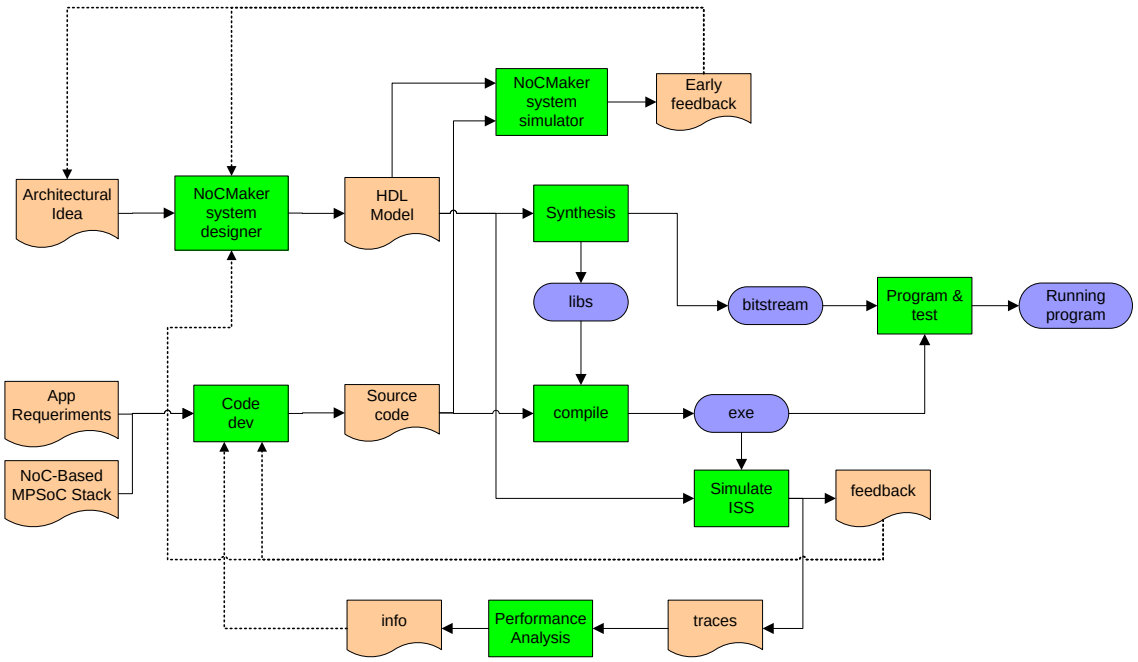


Figure 22 Full methodology flow.

In order to complete our design flow, we provide the developer with several tools and techniques that allow debugging and tracing its application. Once again, NoCMaker gives debugging feedback of the execution of the application at assembly instruction level, and provides tracing files for further analysis [62].

The design flow presented in this work is similar to other existing flows such as D. Atienza in [63]. However, Atienza's work shows a design flow that starts with the

requirements of an application and NoC models (both area and power) and ends with the synthesis of the achieved NoC. This is just a part of the whole process.

Our work completes it by adding two new steps that help the creation and programming of a system by providing software programming tools and performance analysis tools. This completion renders the whole process a systematic flow.

3.2.2. MPSoC System Example

Following the methodology described in previous sections, we built an MPSoC processor targeting a Stratix II EP2S180 Altera FPGA, and we used the software development environment and the explained performance analysis tools to implement, analyze and optimize several applications.

3.2.3. Building Blocks

The design of a complex SoC on FPGAs involves the selection and combination of many possible IP cores such as memories, input and output interfaces, peripherals, processors, etc. These components must be interconnected in order to build up the complete system. In the following subsections, we describe the fundamental blocks used to build our system.

- **Soft-Core Processor**

Altera Nios II microprocessor [64] is a general-purpose RISC processor core, designed under the Harvard architecture paradigm with 32-bit instruction set, capable of offering up to 250 DMIPS. Nios II microprocessor family is composed of three members: Fast Nios II, Economic Nios II, and Standard Nios II, each one optimized for performance,

for minimum logic usage, and for a balance between resources usage and performance, respectively. The whole family uses the same Instruction Set Architecture (ISA), which may be extended to include custom instructions from the programmer.

Nios II family uses Avalon bus [65], which enables multiple data transactions and defines a set of signal types with which a designer can connect any compatible peripheral. Altera provides the designer with SOPC builder, which allows the designer to build their own processor with their number of selected peripherals attached to the Avalon bus. After the selection of components, the configuration of each block feature, and the definition of the memory map, SOPC creates an HDL entity (described either in VHDL or in Verilog) that can be inserted into any design. Due to the resources available in the FPGA platform we are targeting, the number of Nios II processors on the system will be limited to 16.

- ## Floating Point Unit

The Nios II processor does not directly include a FPU. Nevertheless, Altera offers an add-on FPU attached as a Custom Instruction [66]. We use Michael Schoeggl's FPU design [67] instead of Altera's because it supports more arithmetic functions and allows us to determine which floating point functions are synthesized. This allows further optimization possibilities if we need them, be cause FPU is one of the more resource-greedy elements in the system.

- ## Network-on-Chip

NoCs are a necessary part to interconnect the 16 Nios II processor elements. However, there is a huge NoC design space, so a proper choice of the parameters that minimize energy consumption while maximizing performance is not a trivial issue and, in fact, it

could be an arduous task. NoCMaker can generate a fully synthesizable Verilog code, compatible with Altera and Xilinx technologies.

Our MPSoC is composed of one master and 15 slave nodes connected by a 2-D Mesh Network-on-Chip, which is suitable for interconnecting homogeneous cores in a MPSoC due to the simplicity and the predictability of its grid-like structure.

As depicted in Figure 23 (left), the master node is strategically situated in the coordinate (1,1) of the NoC to minimize communication costs with the rest of the nodes. Although there are some differences between nodes, they share some common features. Each node is composed of a Fast Nios II processor with a data cache of 2Kb and an instruction cache of 4Kb. Each processor has an attached custom configurable FPU [67]. Each processor has been provided with a JTAG interface for debug and download of the executable program. The connection of the processor to the NoC is performed via a Network Interface attached to the Avalon bus. The master node assumes the responsibility of orchestrating the work of the slaves. It has access to 16MB of off-chip SDRAM memory through a SDRAM controller attached to its Avalon bus.

In order to ease debug and performance analysis, it also has a UART interface to communicate with an external PC, and a performance counter able to take accurate time measures. The slave nodes are simpler: they have neither UART interface nor performance counter, nor external memory access. Instead, they have been provided with an on-chip memory of 32Kb, which limits the amount of code runtime data that slave nodes can handle.

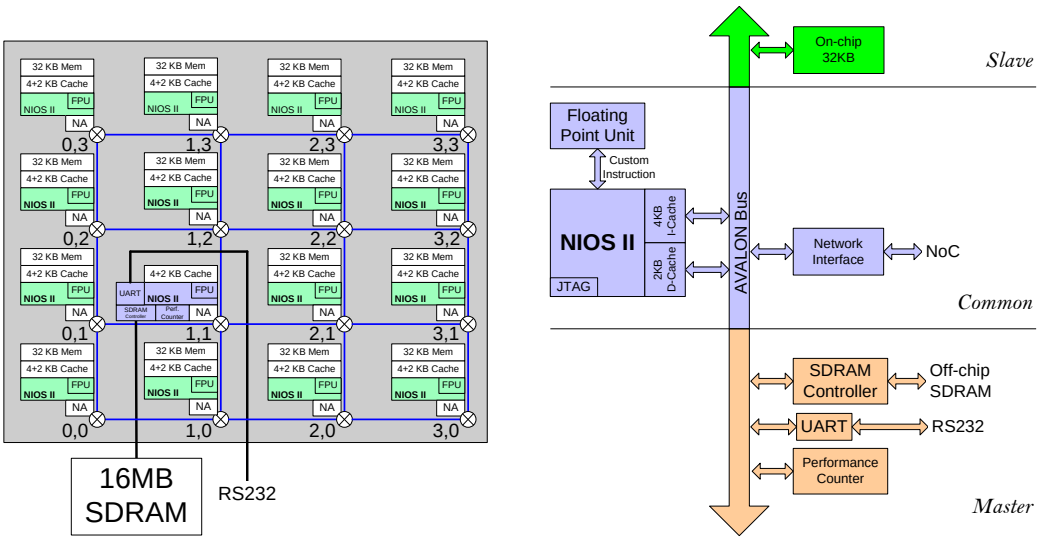


Figure 23 Left) MPSoC based on the 4x4 2-D Mesh NoC. **Right**) Master & Slave node architecture

Figure 23 (right) depicts the similarities and differences in Master and Slave node architectures. One of the important blocks of the design is the Network Interface. The architecture of the NIC is divided into two modules, one focused on the interaction with the processor, or more specifically, with the bus, and the other concerning the interaction with the NoC. The specifications of the Avalon bus determine the architecture of the first module, and the design space point of the NoC determines the second module. Figure 24 (left) shows the architecture of the NIC. To improve the performance of the whole system, the NIC includes a double buffer on the input and output of the NoC. The double buffer allows the processor to inject a packet to the NIC while a previous packet is concurrently being injected into the network. Otherwise, in a single buffer design, the processor should wait until the previous packet is flushed to the NoC before injecting a new packet. The part of the NIC interacting with the NoC must inject and eject flits following the strategies defined by the selected NoC space point.

In our case, we built a 4x4 2-D Mesh NoC that uses wormhole switching and 4-phase handshake flow control. The channel width is 35 bits: 32 bits for flit data, 2 bits for handshake signals and one additional control bit to identify special flits, like header flit

and tail flit. The header flit contains the source and destination address of the packet. The destination address is necessary to be able to route the packet in the network.

In our NoC, we are using distributed routing following XY routing algorithm. The architecture of the router is relatively simple, since no virtual channels are used. As depicted in Figure 24 (right), the router has three main modules: Arbiter module, which computes the route decision for packets going from input ports to proper output ports of the router, and grants the channel required for this routing; Crossbar module, whose unique task is to switch packets from the indicate input port to the output port; and, finally, FIFO modules, whose task is to buffer the packets traveling through the router.

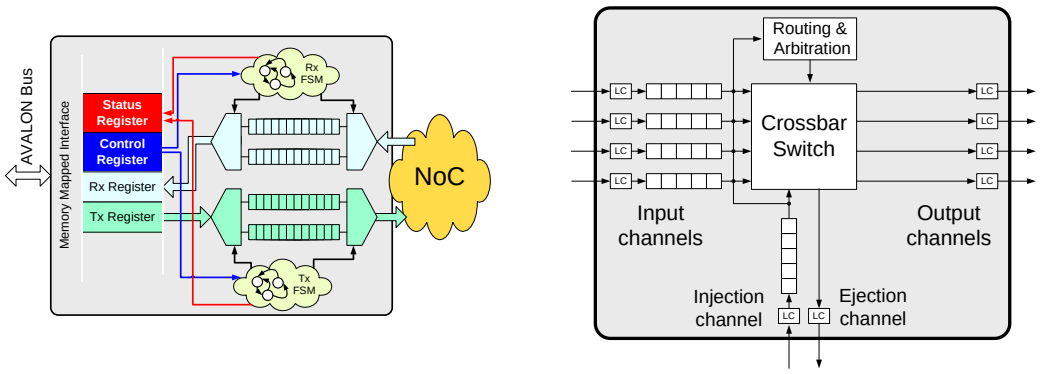


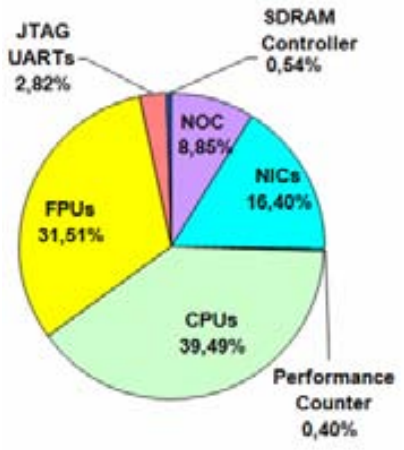
Figure 24 Left) Block diagram of the Network Interface. **Right**) Block diagram of the Router

3.2.4. Synthesis Results

We synthesized and executed our MPSoC design in a Statrix II EP2S180 DSP development board. The FPGA device has 179,400 of LEs and a total on-chip RAM of 9,383,040 bits. Henceforth, we use the word synthesis as a synonym for all the processes involved in producing a configuration bitstream, i.e. synthesis and place-and-route. The synthesis tool reports less than 50% usage of the device's total logic resource capacity and 56% of the total memory usage. The maximum frequency of the circuit is 77 MHz. Figure 25 (right) shows the details of the synthesis results. Almost all memory

resources are devoted to the cache and on-chip memories of the CPUs. Logic resources are more evenly distributed among the different modules of the system as it can be seen in the resource breakdown shown in Figure 25 (left).

The resources devoted to the NoC are very few, just 9%, while the resources devoted to the NICs are larger (more than 16%). The reason for this important cost is that, although the NoC uses a wormhole switching strategy, NICs use a store-and-forward approach, i.e. they store the whole packet in FIFO before injecting it to the network, and use a similar technique for ejecting a received packet. This imposes high requirements on NIC storage. Furthermore, we use double buffering, and the used FIFO design is based on registers instead of memories. When we sum up all these contributions, the consequence is the relatively high logic usage. Also remarkable is the amount of logic used by FPUs, which is comparable to the amount used by CPUs.



Resource Type	Usage
Combinational ALUTs	59648 / 143520 (42%)
Dedicated logic registers	66332 / 143520 (46%)
Memory bits	5289152 / 9383040 (56%)
DSP block	704 / 768 (92%)

Figure 25 Left) Resource utilization breakdown. **Right**) Synthesis Results

A possible conclusion from this observation is that applications that do not require floating point operations could reuse FPU logic to double the number of CPUs. However, one has to take into account that increasing the number of CPUs would raise

the size of the NOC, the resources used by NICs, and memory demands, which are already over 50%.

3.2.5. Scalability Test

For the evaluation of the scalability of the platform, several real life applications were carried out:

The first application implemented was a matrix multiplication (MM) which has many important scientific and engineering applications. In order to parallelize MM operation, we used Sub-Matrix Multiplication method that subdivides the original matrix into multiple sub-matrices of 30x30 elements.

The second application was like the first one but working with sub-matrices of 50x50 elements.

The third application applies the Hough transform [69] to detect circles, which is a common application used in computer vision and in digital image processing. The Hough transform may be used to determine the parameters of a circle when a number of points falling on the perimeter are known. The goal is to find the center of the circle (a,b), which can be described with the parametric equations $X = a + R*\cos(\theta)$ and $Y = b + R*\cos(\theta)$, where R is the known radius and θ is the angle that sweeps through the full 360-degree range. The geometric position of (a,b) in the parameter space falls on a circle of radius R centered at (X,Y). The center point will be common to all parameter circles and can be found with a Hough accumulation array (see Figure 26).

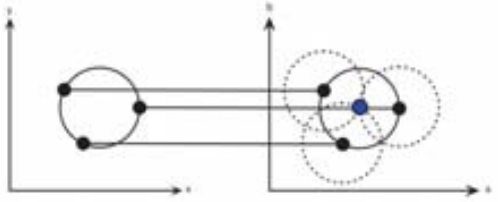


Figure 26 Hough Transform to detect circles where each point in geometric space (left) generates a circle in parameter space (right).

We implemented the transform to detect circles of fixed radius 10 in an original monochrome image of 640x480 pixels. The applications have different communication vs. computation ratios. The communication vs. computation ratio of the first application is higher than the ratio in the second, and the second is higher than the third. This means that in the circle detection application the number of operations that each node has to execute is higher than the number of bytes that must be transferred.

At the beginning of each benchmark, initial time t1 is measured by ocMPI_Wtime, and, after the execution of ocMPI_finalize, time t2 is measured and we can calculate time Tp, the difference between t1 and t2. In order to calculate the speedup obtained.

Time Tp is compared with the time spent for just one processor to perform the serial version of the application, Ts. Therefore, speedup is calculated by Ts / Tp. The results of scalability analysis are shown in Figure 27.

It is clear that the lower the communication vs. computation ratio, the more speedup gained in the multi-core platform. It is remarkable that a speedup factor of 12 using 16 processors has been obtained in the circle detection application. As it was commented previously, within an ideal system, we could expect a maximum speedup factor of (Number of Cores – 1), because the master node is only dispatching the workloads among worker slaves. It is interesting to note that MM applications do not scale to more than 5 and 7 processors, while obtaining a modest speedup factor of 2.5 and 4 respectively.

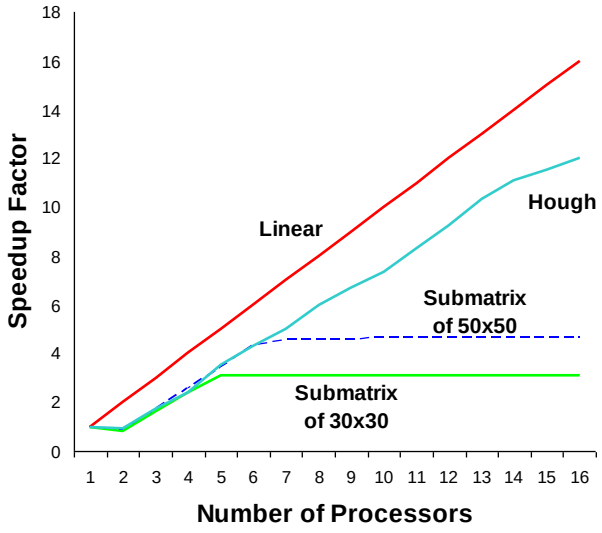


Figure 27 Scalability

These results are obtained due to the limits in on-Chip memory of the small FPGA device targeted. The results would be better using devices with more internal memory. In any case, when the scenario rules out obtaining better results, we should either invest the idle processors in other tasks, or invest their logic resources in more logic circuits that can help us improve the performance.

3.3. Conclusions

In this chapter, a complete systematic design flow to develop both NoC-based MPSoC systems and parallel applications is shown. This methodology is not limited to restrictive structures neither to restrictive interconnection systems like other similar frameworks are. We combined commercial tools like SoPC builder from Altera with open source tools like NoCMaker to develop and design the whole system and to analyze the performance of the parallel program. The stack includes transport and driver device layers that make the network specifications transparent to the software developer. In addition, we developed and applied a combination of techniques to

analyze and, when possible, diagnose the reasons for performance problems. We proved that the building of many-{soft}-cores on FPGAs is a feasible and attainable possibility. A NoC-based MPSoC system with 16 processors was built and it was proven effective to scale some simple applications. The results of this chapter produced a journal article [102].

4. MPI Implementations

4.1. Overview

Once a NoC is selected, and the MPSoC is synthesized, the software developer needs tools to program it. The parallelization process usually starts with a serial application that is analyzed, either by hand or semi-automatically, to detect the regions of code containing potential parallelism. In this work, we propose the use of virtual prototyping in combination with trace generation to analyze the serial code and to help the detection of such potential parallelizable regions.

Additionally, it is necessary to know which programming model paradigm is to be followed to develop the parallel application. In this field, two main parallel programming models have been implemented and tested on many-core MPSoC architectures: OpenMP [28], for shared memory architectures, and MPI [70][71], for message passing architectures. Due to the inherent distributed nature of NoC-based many-core systems, message passing architectures and programming models (e.g. MPI) could overcome the non-determinism and the scalability limits that cache coherence protocols introduce in shared memory architectures. Other reasons to support MPI vs. alternative Application Programming Interfaces (API), like Multicore Association Communication API (MCAPI [72]), are that MPI is a very well-know API and parallel programming model, and debug and trace tools are currently available [73][74]. The drawback of using MPI is that it requires more programming effort as it is a communication protocol in which messages must be programmed explicitly by the developer, unlike OpenMP, which is an easier way of parallelizing that uses compiler directives and the compiler manages the threads needed for the code to work in parallel.

Once the programming model has been chosen, the programmer needs an environment to create applications. FPGA manufacturers such as Altera or Xilinx provide a soft-core design environment for software development. Altera owns its IDE (Integrated Development Environment), and Xilinx has SDK (Software Development Kit). Both environments are based on the familiar Eclipse framework and make use of the GNU gcc C/C++ compilation chain to build executables.

When creating multiple CPUs, FPGA manufacturers' tools for building systems such as Altera SOPC builder create a separate software library for each processor, because each processor can be architecturally different. Differences could be related to cache sizes, FPU (Floating Point Unit) availability, attached devices or even the extension of the instruction set. To support this heterogeneity, even if there is a single program for all the nodes, the source code must be compiled against each processor platform, thus producing different executables. Figure 28 shows, for Altera case, the process previously explained.

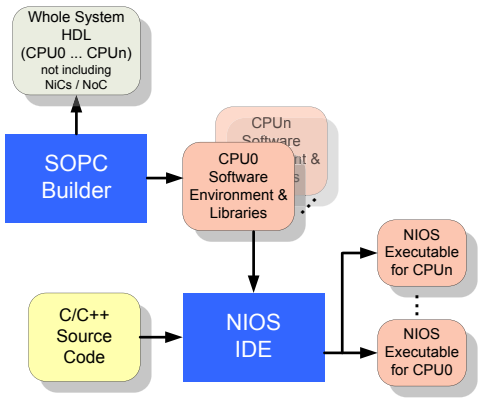


Figure 28 Development process

In order to make the programming NoC-based MPSoC systems easier, we provide a lightweight version of MPI to implement applications on top of the architecture. Our particular MPSoC architecture, described in section 4, favors the adoption of a message passing programming model over other alternatives, such as shared memory

programming models. Our MPI stack (ocMPI [75][76]) does not require any operating system, or any running daemon (e.g. mpirun). However, it requires some software layers below the ocMPI API to provide the communication primitives needed for efficient communication.

Figure 29 shows a comparison of the classical OSI stack against our MPSoC stack. We use a transport protocol and a device driver to communicate with the Network Interface Controller (NIC) device through its memory-mapped interface. Unlike a traditional communication stack, we lack a network layer, because its functions are essentially embedded in the NoC. The device driver layer implements the low-level instructions the processor needs to access NIC registers through the Avalon bus. The main functions of the driver's top layer are *send* and *receive*. *Send* function has three parameters: (i) a pointer to the data to send, (ii) the length of the buffer, and (iii) the target address where the data must be sent. When the *send* function is called, it prepares a packet following the characteristics of the NoC and instructs the correspondent finite state machine in the device to start injecting the packet into the NoC. *Receive* function has two parameters: (i) a pointer to a buffer where received data will be stored, and (ii) a pointer where the length of received data will be indicated. Driver functions are basically blocking, so there are some useful functions to check the status of the device, like detecting whether data is available at the reception buffer.

In order to provide essential features like unblocking primitives, fragmentation and reassembly, and channel multiplexing, we developed a lightweight transport layer. This is especially important at receiving endpoints. At reception time, the transport layer classifies incoming data from the network by packet's source address. If the target node is specifically waiting for a transmission from the source node, incoming data are transmitted to upper layers. However, when packets are received from undesired

sources, they are stor ed in a re ception buff er that c an late r be c hecked whe n the application calls a receive function again.

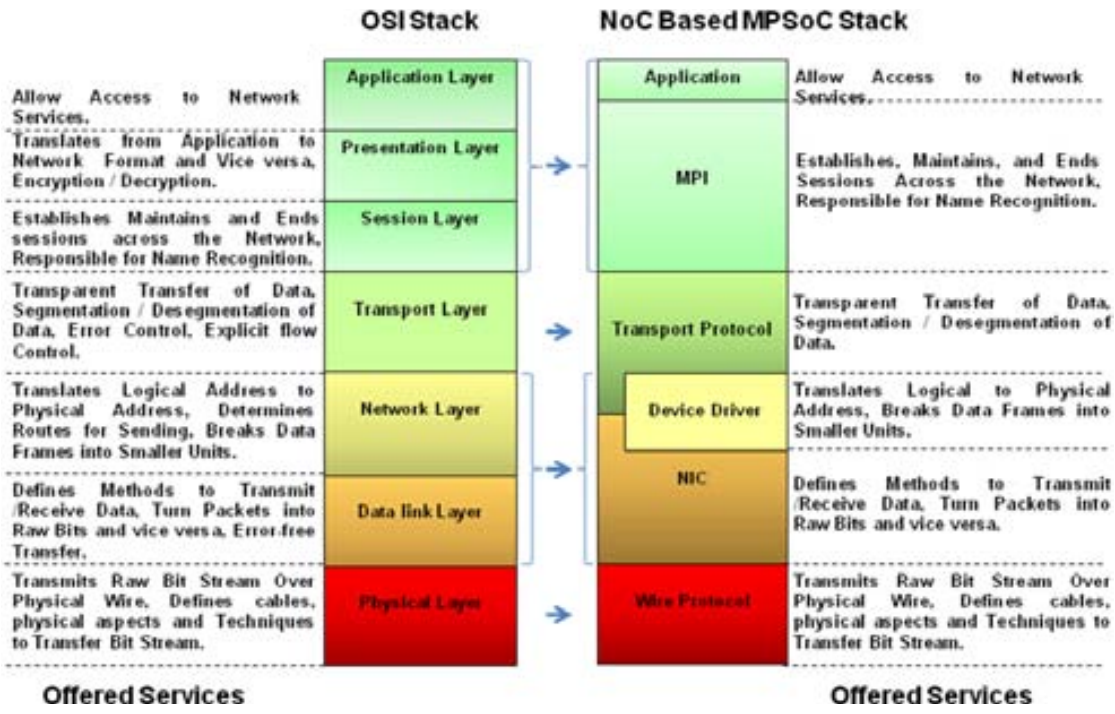


Figure 29 Traditional OSI stack vs. MPSoC software stack

The upper layer of our communication stack is the ocMPI API, which is a minimal subset of MPI API. ocMPI library supports up to 11 standard MPI functions. Table 1 (left) shows some of the most relevant functions.

Function	Description
ocMPI_Init	Initializes ocMPI execution environment
ocMPI_Finalize	Terminates ocMPI execution environment
ocMPI_Comm_rank	Determines the rank of the calling process in the communicator
ocMPI_Comm_size	Determines the size of the group associated with a communicator
ocMPI_Send	Performs a basic send (blocking send)
ocMPI_Recv	Performs a basic receive (blocking receive)
ocMPI_Wtime	Returns an elapsed time on the calling processor

Layer	Memory Size
ocMPI	11 Kb
Transport	2 Kb
Device Driver	1 Kb

Table 1 Left) Implemented ocMPI functions **Right)** Memory footprint of software stack

The footprint of software layers, Table 1 (right), is minimal and requires 14Kb of memory.

Table 2 shows a footprint comparison between existing MPI alternatives for embedded systems.

	OpenMPI [78][79]	**MPICH [80]**	**TMD-MPI [81]**
Availability	Open Source	Open Source	Proprietary
MPI library size	25MB	7MB	9Kb
All layers size	40Mb	47Mb	--
MPI commands supported	300	300	11
	SoC-MPI [82]	**RAMPSoC-MPI [77]**	**ocMPI [75][76]**
Availability	Proprietary	Proprietary	Open Source
MPI library size	11-16 Kb	37 Kb	11Kb
All layers size	--	43Kb	14Kb
MPI commands supported	jun-18	18	11

Table 2 ocMPI compared with different MPI implementations for embedded systems.

Message Passing Interface is the standard de facto used in distributed memory systems, like HPC clusters, for communication among processors. MPI promotes data locality, which usually goes in favor of scalability. MPI is a real option to program highly parallel and scalable many-soft-cores. Furthermore, the portability and extensibility of MPI API make it easy to be tailored to many-soft-cores, and MPI API is a very well-known programming model for the programmer community.

ocMPI has been developed in ANSI C in order to minimize the footprint of the library which can be compiled by many processors, like NIOS II, MicroBlaze, ARM for instance, that support gcc-like tool chain. On ocMPI a minimal selection of standard MPI functions are selected.

Table 3 shows the main functions of a minimal working configuration of ocMPI.

Function	Description
MPI_Init	Initializes MPI execution environment
MPI_Finalize	Terminates MPI execution environment
MPI_Comm_rank	Determines the rank of the calling process in the communicator
MPI_Comm_size	Determines the size of the group associated with a communicator
MPI_Send	Performs a basic send (blocking send)
MPI_Recv	Performs a basic receive (blocking receive)
MPI_Wtime	Returns the time on the calling processor

Table 3 Minimal set of functions of ocMPI

Using these functions, many MPI applications can be developed and other more complex MPI functions (like collective communication primitives) can be implemented by invoking these simple ones.

MPI_Init and MPI_Finalize are management primitives. MPI_Init, sets up the MPI environment and, as in the homonymous standard MPI function, any other ocMPI function can appear previously, and MPI_Finalize finalizes the execution environment and any other ocMPI function can appear after MPI_Finalize is called. Additionally, MPI_Comm_Size and MPI_Comm_rank are management primitives. The relevance of those functions is explained in the following sections. MPI_Send and MPI_recv are point-to-point communication primitives that implement the basic blocking send/receive primitives. Finally, MPI_Wtime primitive can be used for time measurement.

4.2. Shared Memory

Several options appear when trying to implement ocMPI over shared memory architectures. However there are two main options: (i) implement a single queue where the communication must happen, or (ii) create several queues.

When a single queue is created the writers leave the messages on the first free position available on the queue and, when *receive* is posted, the reader analyzes the headers of the messages to find the whished message. This implementation can be an option when the application performed is not intensive in communication or in messages.

When implementing several queues, three options appear. The first one consists in implementing a single queue for each receiver. In this case, the senders access to the specific queue of the receiver to leave the message, and it is the duty of the receiver to analyze the queue and find the message wanted. In the case of receiving from ANY_SOURCE the complexity is similar because it is the same queue that has to be analyzed.

The second option is to implement a queue in the senders. Using that solution, the sender writes always on the same queue and it is the receiver that access different queues depending on the source of the message, and also, searches for the wanted message or messages. In the case of receiving from ANY_SOURCE the receiver has to analyze all the queues from all the senders, which makes it quite an impractical option.

Finally, the third option is to create a matrix of queues where each couple of sender/receiver has a dedicated queue. In that solution, it is not necessary to analyze the messages to find the wanted one/ones, since the use of such queues is deterministic and, therefore, all the messages inside a queue have the same source and destination. In the case of receiving from ANY_SOURCE the receiver has to analyze just the queues of the matrix where it is the destination.

Regarding the method chosen, all the queues must be accessed from different points at the same time. That implies that all the queues must be synchronized and protected.

That can be done using either software or hardware solutions. For software solutions a mutual exclusion algorithm can be implemented, such as a Lamport's bakery-based algorithm. For hardware solut ions, a ha rdware mut ex c an be used. N IOS-II tool s already include a Hardware mutex IP-Core.

In addition, the implementer has to decide what to post to the queues. Either pointer to messages or complete messages. The oc MPI implementation only p ost messages pointers into the queues, a nd b ypasses the da ta c ache b y using dire ct I/O processor instructions such as *ldwio* and *stwio*.

Two are th e ve rsions of the MPI li brary im plemented ove r two diff erent shared memory architectures.

- Recore System (Xentium + Leon)
- STHORM (P2012 platform)

4.2.1. STHORM

Within t he C ATRENE European p roject COBRA [83], it a ppeared the possibility to work with one of the most relevant massively p arallel programmable pr ocessor. The STHORM [84] platform oriented to be an area and power efficient many-core system.

The STHORM fabric is highly modul ar and sc alable, since it is based on a c luster-like a rchitecture with independent power and c lock domains. Figure 30 shows the architecture of the STHORM platform. It can be se en that the clusters are interconnected using a fully-asynchronous-NoC (ANOC). Each cluster has a copy of the ENcore pr ocessor, which is a 16 p rocessors core with independent instruction str eams, and the Cluster controller (CC) processor that in cludes a DMA sub-system and se veral interfaces to connect the global f abric ANOC, the local asynchronous ne twork for

extend the fabric with accelerators. The CC processor is in charge of booting and initialising ENcore system.

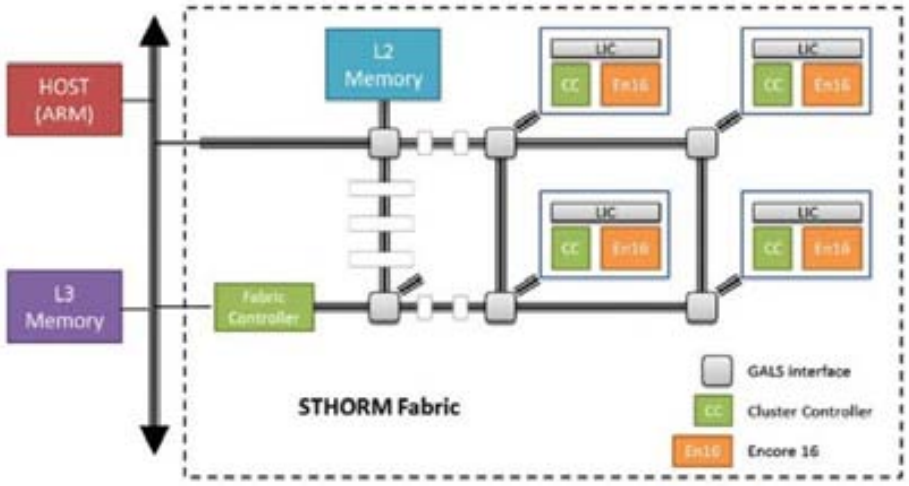


Figure 30 STHORM architecture template [95][96]...

Once the access to the platform simulator was granted, a study process of the platform was performed to identify its particularities and, therefore, to adapt the requirements of the library to the platform.

Since the platform is based on shared memory architecture and MPI libraries where initially implemented to be used with distributed memory architectures, we identify 3 main points that must be fulfilled in order to adopt an MPI-like programming model:

1. BOOT. We need to know how to ensure that each node on the system loads the same code at the starting time.

   This will allow us to use some of the management MPI function concretely:

   a. MPI_INIT (). Initializes the MPI software library.

   b. MPI_FINALIZE (). Ends the MPI library.

2. PROCESSOR ELEMENT IDENTIFICATION. It is necessary to know the size of the system; this is the number of processor that composes such system.

   This knowledge will allow us to use the MPI functions;

   a. MPI_RANK (). Gets the rank of a process in the MPI software library.

b. MPI_SIZE (). Gets the number of all concurrent processes that run over the multiprocessor system.

3. SHARING DATA. The main functionality of a message passing system must be sharing data. Such functionality is implemented basically with the MPI functions;
    a. MPI_SEND (). Send function.
    b. MPI_RECEIVE (). Receive function.

How to synchronize and do copies of the messages are the two points to satisfy in order to successfully implement these sharing data functions. In a hardware approach, the copy could be implemented using Direct Memory Access (DMA) controllers. In a software approach by using Memcopy instructions.

To achieve synchronization between the message sending and receive, we could make use of the specific hardware existing in the ST platform; Mailboxes and Queues. Other option to synchronize is using the shared memory resource and implementing a TargetXSource 2D-matrix array that will indicate if there is any message waiting to be received. This matrix will be accessed using polling technique.

These six functions are the minim required to design the message passing communications for any application.

Following these premises, a first version of the MPI library for ST platform was developed. In this version, the critical points and the 6 MPI functions commented were implemented. The library used some of the high-level tools available on the ST platform, such as the PPP (Parallel Programming Patterns), which provides high-level solutions to synchronize, communicate and share data.

This first library, the platform was fixed to a static number of nodes N using the ADL (Architecture Description Language), and it has been described as an NxN matrix of full-connected communication channels.

These channels are implemented as queues using the simpleIterator pattern, and the required interfaces to use the queues are managed using queueIteratorWriter and QueueIteratorReader.

The initial library was being updating year by year with the newest facilities that the ST platform was offering. The runtime of the latest versions of the MPI library relies on the services of the HAL layer available on the STHORM platform.

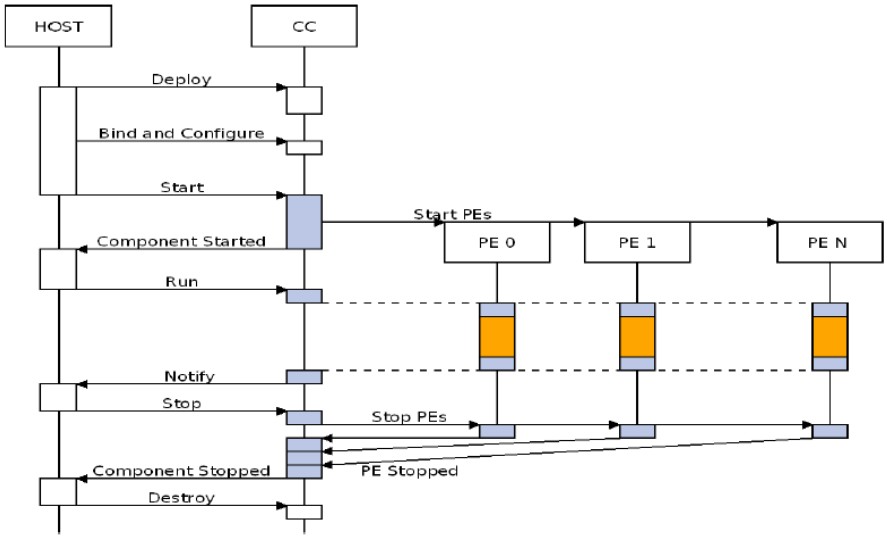


Figure 31 Developed execution engine on top of the base runtime services and the HAL.

The Figure 31 shows the boot mechanism of the library. Once the host processor deploys the cluster image of the application (that includes the MPI library) to the cluster controller and starts the engine, the cluster controller initialises the PE elements using the MPI library.

Once the system is initialised with the MPI library, the PE cores can share information using the MPI_SEND and MPI_RECV primitives.

At the end of the application, MPI_FINALIZE is called by all the computing elements, and the cluster controllers finally end the MPI environment as well as the cluster execution and notify that to the host that destroys the image.

Additionally, a trace generation library was developed to support the MPI library. The traces generated by this library use the OTF format (Open Trace Format), a trace definition used in big parallel computer platforms in the HPC world. Using such library, the user obtains traces of each process element involved in a specific computing task. Therefore, using this format, the user can analyse the performance of the system and the MPI library. The files that the trace library generates can be visualised by tracing visualisation tools, such as Vampir, TAU or Paaver.

One of the test benches used to validate the MPI library was the Mandelbort set calculation using a fine grain methodology that computes pixel by pixel, as shown in Figure 32 and Figure 33.

```
MPI_Init(0, NULL);
MPI_Comm_rank(0, &rank);
MPI_Comm_size(0, &size);
if ((rank == 0))  {          // MASTER CODE
          …
          for (y=ymin; y<=ymax; y+= (ymax-ymin)/divy) {
             for (x=xmin; x<=xmax; x+= (xmax-xmin)/divx) {
               dst = slave+1;
               if(dst]) {
                  MPI_Recv(&res, 1, MPI_FLOAT, dst, 0, MPI_COMM_WORLD, &status);
                  … // store result
               }
                   vals[0] = x;
               vals[1] = y;
                   MPI_Send(vals, 2, MPI_FLOAT, slave+1, 0, MPI_COMM_WORLD);
                    slave = (slave + 1) % slaves;
            }
          }
          vals[0] = -10; vals[1] = -10;          // special values to force exit of slaves
          for (i=1; i < size; i++)
      MPI_Send(vals, 2, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
}
else {    // SLAVES
                    while (1) {
              …
                        i++;
                        int ret = MPI_Recv(&vals, 2, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
                              &status);
                        float x = vals[0];
```

```
                    float y = vals[1];
                    if ((x <= -10) && (y <= -10))  return 0;
            v[0] = computePoint(x,y);
                    MPI_Send(v, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
                }
        }
int computePoint(float x0, float y0) {
          …
          x = x0; // x co-ordinate of pixel
          y = y0; // y co-ordinate of pixel
          float dos2 = 2.0 * 2.0;
          while ( (x*x + y*y < dos2)  &&  (iteration < max_iteration) )    {
                  xtemp = x*x - y*y + x0;
                  ytemp = dos*x*y + y0;
                  x = xtemp;
                  y = ytemp;
                  iteration++;
             }
          if (iteration == 0)
                    colour = 1;
      else if ( iteration == max_iteration )
                  colour = max_iteration;
          else
                colour = iteration;
   return  iteration;
}
```

Figure 32 MPI parallel code for Mandelbrot set calculation executed using STHORM platform.

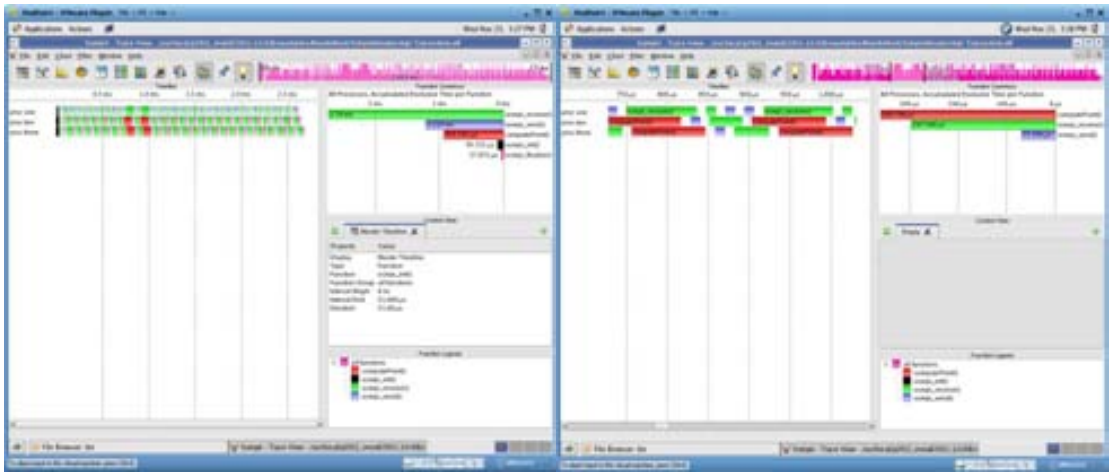


Figure 33 Visualisation of the traces obtained when performing Mandelbrot set calculation with 3 processors over STHORM platform. Right image zooms in a small section of the left image.

Additionally, in the same project the MPI library was used to adapt the code of some parts of the Ecomunicat's [100] application of people counter. The Figure 34

shows the application d ata flow a nd, in Figure 35 the a pplication data flow in the parallel version.

Figure 34 People counter application flow (serial version).

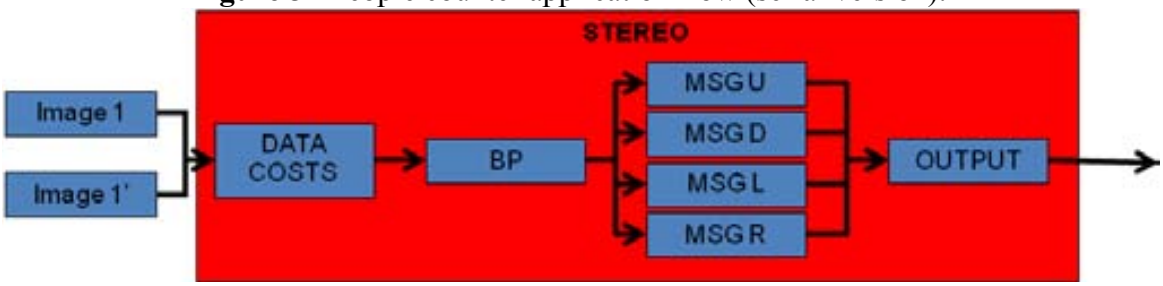


Figure 35 Parallel version of 3D generation code.

Following these lines, in Figure 36, we can see the code used, and Figure 37 shows some traces obtained from the execution of the application over the STORM platform.

```
MPI_Init(0, NULL);
MPI_Comm_rank(0, &rank);
MPI_Comm_size(0, &size);
if ((rank == S0)) { // MASTER CODE
…
for (int t = 0; t < ITER; t++) {
      for (int y = 1; y < height-1; y++) {
           for (int x = ((y+t) % 2) + 1; x < width-1; x+=2) {
                send(u,x,y+1,S1); send(l,x+1,y,S1); send(r,x-1,y,S1); send(data,x,y,S1); //MSG U
                send(d,x,y-1,S2); send(l,x+1,y,S2); send(r,x-1,y,S2); send(data,x,y,S2); //MSG D
                send(u,x,y+1,S3); send(d,x,y-1,S3); send(r,x-1,y,S3); send(data,x,y,S3); //MSG R
                send(u,x,y+1,S4); send(d,x,y-1,S4); send(l,x+1,y,S4); send(data,x,y,S3); //MSG L
                 …
           }
      }
 }
 …
 }else { // SLAVES
    while (1) {
          …
          i++;
          MPI_Recv(a1, VALUES, MPI_FLOAT, S0, 0, MPI_COMM_WORLD, &status);
          MPI_Recv(a2, VALUES, MPI_FLOAT, S0, 0, MPI_COMM_WORLD, &status);
          MPI_Recv(a3, VALUES, MPI_FLOAT, S0, 0, MPI_COMM_WORLD, &status);
          MPI_Recv(a4, VALUES, MPI_FLOAT, S0, 0, MPI_COMM_WORLD, &status);
          compute_msg(a1,a2,a3,a4,dst);
          MPI_Send(dst, VALUES, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
      }
}
…
```

Figure 36 MPI Parallel code used on 3D generation code.

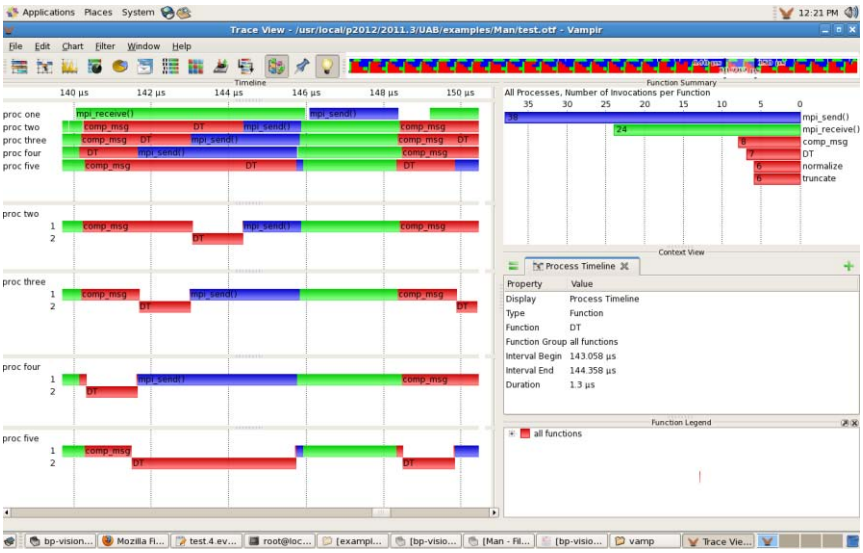


Figure 37 Traces obtained when executing Ecomunicat's application of people counter.

4.2.2. Recore System

Within the 8th HIPEAC call for industrial PhD internship, appeared the possibility to work with Recore Systems at Enschede, The Netherlands, on "Reconfigurable multi-core SoC programming". Recore Systems participates contributes to a significant number of Dutch and Europe-wide research projects. Recore Systems collaborates with industry partners and highly-regarded research institutes on advanced research challenges in the areas of multi-core programming and reconfigurability.

In this context, Recore Systems wanted to investigate programming of multi-core SoC architectures using Recore's multi-core systems, and, therefore, the focus of the main activities done at Recore was to implement a subset of the standard MPI library for a multi-core system based on Xentium [85] and NoC technology.

The work done was divided in some steps:

- Study Recore's technology and its specific characteristics. The main activities during this phase were reading all the specifications of the Recore technology and meeting with Recore's engineers almost once per day.
- Make a proposal of MPI primitives to implement over a shared-memory heterogeneous multi-core processor that includes two Xentium DSP processors, one LEON2 processor, several memory tiles…, and make a propose of how to implement that primitives.
- Test the library on an FPGA prototype and on a simulator.

The starting point was to create a layered software stack to abstract the details of the hardware from layer to layer, and build the MPI layer on top of the software stack enabling the user of the system to program the platform while keeping hidden unnecessary details. Figure 38 shows the software stack created for Recore Systems to be used with MPI library.

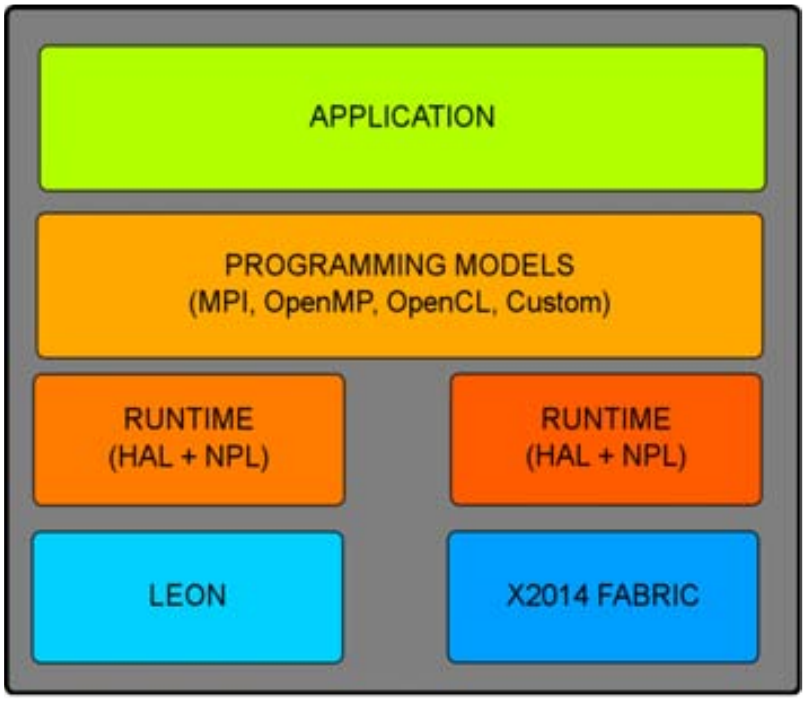


Figure 38 Recore's Software stack diagram.

X2014 is a multi-core SoC with two main components: (i) a single main core LEON2 processor, and (ii) a NoC fabric as a co-processor for the main processor. This system allows an application to run on the main processor and to accelerate intensive computing by deporting it on the computing fabric.

Therefore, It was assumed that the main processor is a single core LEON2 that runs without any operating system (right now), and (nowadays) a homogeneous fabric based on Xentium VLIW processors (but prepared for heterogeneity).

The software stack is split into 2 layers:

- The runtime layer. This layer provides low-level services that are common to higher-level layers. It provides for example services for fabric code loading, or memory transfers. It contains 2 modules:
    - Hardware Description Layer (HAL). The hardware description layer associates a symbol to certain specific hardware elements, and contains the map of the hardware. As example:
        - Map of the hardware (Figure 39)

```
#ifdef XENTIUMCC
...
#define MAILBOX_BASE 0x80000
…
#endif
```

Figure 39 Recore's hardware mapping example from HAL library.

        - Association of hardware elements (
        - Figure **40**)

```
typedef struct XentiumRegs
{
        /* Status bits + control registers */
        volatile unsigned int mlbx[4];
        volatile unsigned int signal[8];
        volatile unsigned int dummy2[2];
        volatile unsigned int timer[2];
        volatile unsigned int irq;
        volatile unsigned int reset;
  } XentiumRegs;
```

Figure 40 Recore's hardware assiciation example from HAL library.

- Native Programming Interface. The native programming layer is a low-level API which provides the "most efficient" use of the system resources based on HAL, and therefore giving more level of abstraction.

- The programming model layer. This layer provides high-level environments for parallelizing applications. For example MPI.

Based on the Runtime layer, Recore could create its own programming model that gives the better way of parallelizing application for your system.The basic implementation was a subset of the standard MPI library with seven functions including: *mpi_init* (Figure 41), *mpi_finalize*, *mpi_comm_size*, *mpi_comm_rank*, *mpi_send* (Figure 44). *mpi_recv* (Figure 45) and *mpi_wtime*. A posteriori, the synchronous *send* (Figure 42) and *receive* (Figure 43) primitives were added.

4.2.3. Communication Mechanism

Once 1 Master and N slaves on the system are defined (typically, the Leon2 processor will be the Master and Xentium processors, Slaves), the implemented communication mechanism is:

MPI_Init

0. Master: Resets:
1. Clk NoC, Clk Xentiums, IRQ, Timer, Communication addresses
2. Master: Boots Xentiums processors.
3. Master: Defines ID of each processor.
4. Slaves: Reset:
5. Timers

Figure 41 Recore's MPI_Init process.

MPI_SSend (for Synchronous Send),

Figure **46**, Figure 47, and Figure 49.

0. Prepare message header

1. Access_to_SRC_Position_at_DST_Communication_Memory_(Shared_MT1)(Set 1)
2. Wait_for_Response_at_(Shared_MT1)
3. Tranfer_Data_to_@_obtained_from_2
4. Access_to_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (Clear)
5. Wait_for_Response_at_(Shared_MT1) (==0)
6. Return

Figure 42 Recore's MPI_SSend steps.

MPI_SRecv (for Synchronous Recv),

Figure **46**, Figure 47, and Figure 49

0. Wait_for_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (!= 0)
1. Access_to_Response_at_(Shared_MT1) (Set @ to Recv Data)
2. Wait_for_Clear_at_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (==0)
3. Get_Data
4. Access_to_Response_at_(Shared_MT1) (Clear)
5. Return

Figure 43 Recore's MPI_SRecv steps.

MPI_Send (for Blocking Send),

Figure **46**, Figure 48, and Figure 50.

0. Prepare message header
1. Access_to_SRC_Position_at_DST_Communication_Memory_(Shared_MT1)(Set 1)
2. Wait_for_Response_at_(Shared_MT1)
3. Tranfer_Data_to_@_obtained_from_2
4. Access_to_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (Clear)
5. Wait_for_Response_at_(Shared_MT1) (==0)
6. Return

Figure 44 Recore's MPI_Send steps.

MPI_Recv (for Blocking Recv),

Figure **46**, Figure 48, and Figure 50..

0. Wait_for_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (!= 0)
1. Access_to_Response_at_(Shared_MT1) (Set @ to Recv Data)
2. Wait_for_Clear_at_SRC_Position_at_DST_Communication_Memory_(Shared_MT1) (==0)
3. Get_Data
4. Return

Figure 45 Recore's MPI_Recv steps.

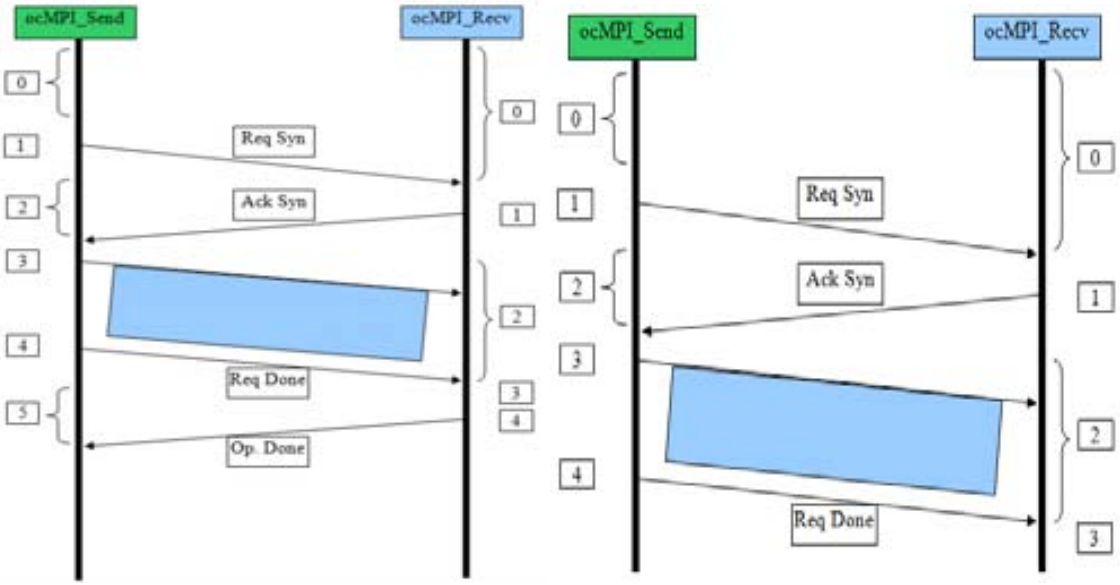


Figure 46 Left) Blocking Send/Recv rendezvous. **Right**) Synchronous Send/Recv rendezvous

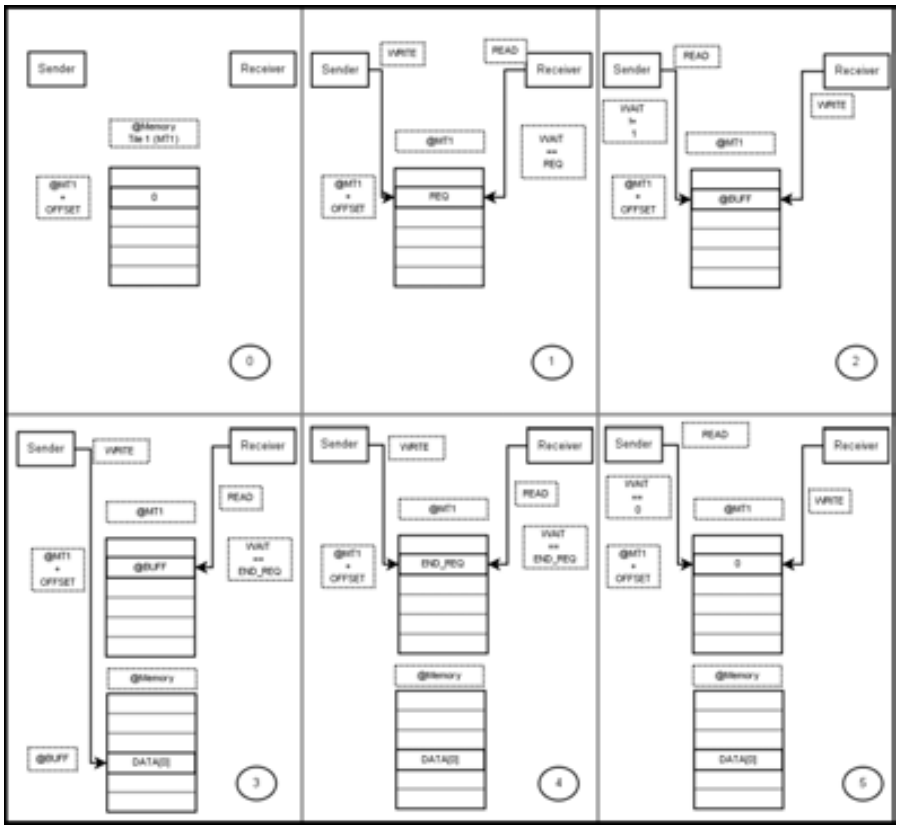


Figure 47 Synchronous Send/Recv implementation with centralized memory.

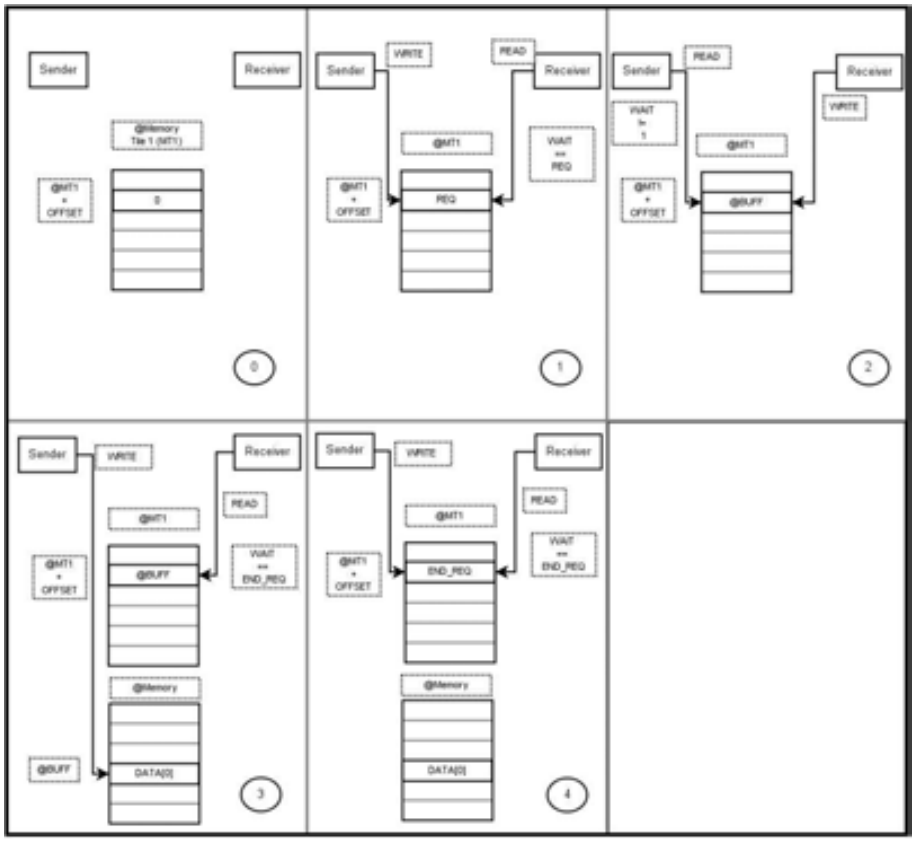


Figure 48 Blocking Send/Recv implementation with centralized memories.

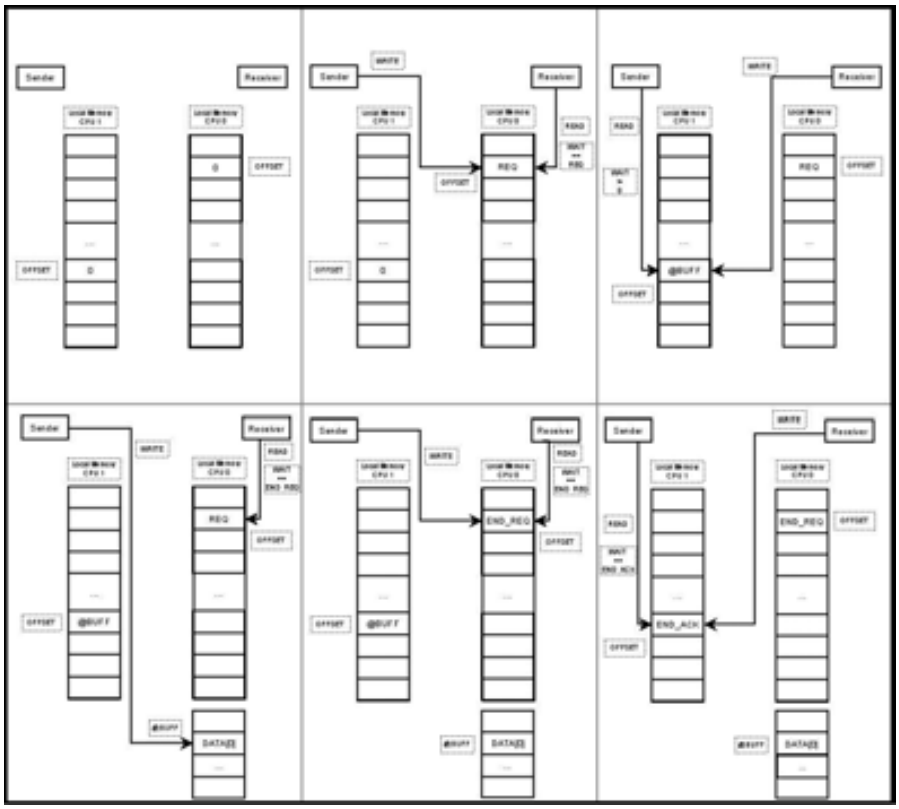


Figure 49 Synchronous Send/Recv implementation with distributed memory.

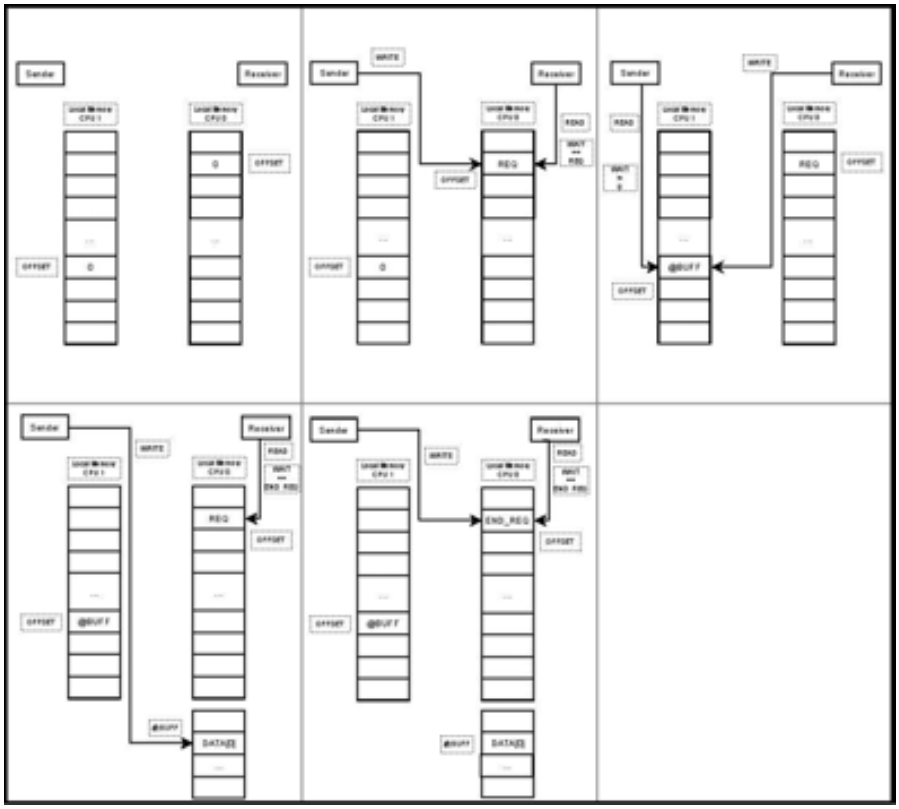


Figure 50 Blocking Send/Recv implementation with distributed memory.

The overall footprint of the library was of 43Kb including all the layers of the software stack.

Once that version was probed to work on a real FPGA-based multi-core system, the library was enhanced including different types of *mpi_send*: blocking send/receive, non-blocking send/receive, and synchronous send/receive. Moreover, different delivery protocols: Rendezvous and eager were included.

Also, a new multi-core shared-memory system was designed and tested on a FPGA board including 6 Xentium processor, 1 LEON2 and 4 memory tiles. Finally, also that system was programmed using the MPI library implemented.

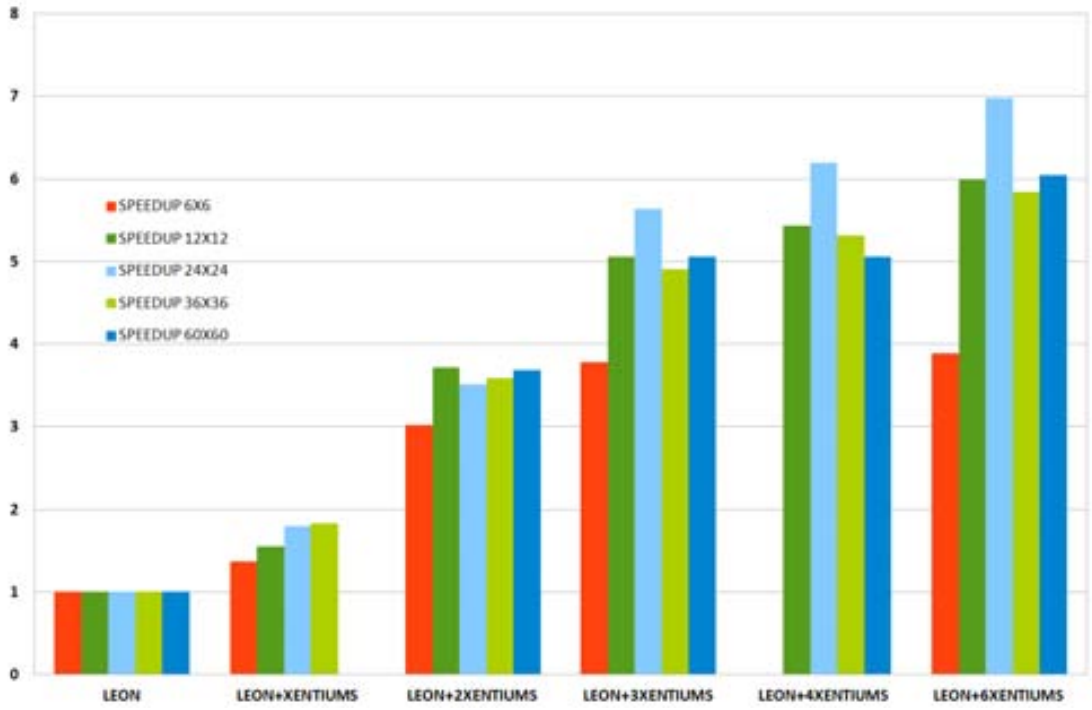


Figure 51 Parallel MPI matrix multiplication execution over Recore's platform

Figure 51 shows the results when running several executions of the matrix multiplication operation with distinct matrix sizes. The algorithm used was the simplest multiplication of column X row, and it was parallelised sending to each computing processor columns and rows.

At the end of the work, two real multi-core systems fully programmable using MPI were achieved, as well as the possibility to use the MPI library on Recore's Xentium simulator.

4.3. Distributed memory

For distributed memory architectures the implementation of MPI_Send and MPI_Recv functions will use the network to emit messages. Therefore, it is not necessary to implement any additional communication mechanism as it happens for shared memory architectures.

However, it is necessary to implement a delivery protocol to synchronize the sender and the receiver. This may be expressed as a choice between two different options: using eager protocol or using rendezvous protocol [86].

In eager protocol, the sender sends messages regardless of the state of the receiver. Therefore, implementing eager protocol requires also the implementation of a software transport layer at the receiver point to be able to multiplex several streams from several sources.

Eager protocol could achieve better results in terms of performance in some applications. However, since messages are sent regardless of the state of the receiver, incoming messages whose MPI_Recv has not been invoked must be stored within a buffer at the destination until a receive call is posted by the MPI application. Such situation becomes worse in a multi-point communication scenario where this protocol has a great probability of going out of order at the delivery of the messages emitted from several sources. This behaviour implies the need of several buffers in the destination nodes to order the messages coming from each source.

The need of buffers in the reception nodes creates an overhead at managing the memory copies, which implies penalizations in execution time due to these copies, and in extra memory space request for the implementation of such buffers, which are a serious handicap for the usual lack of memory resources of distributed many-soft-core systems.

In rendezvous protocol, there is a global flow control synchronizing all the system. The use of this protocol solves the problem of needing extra buffer space to copy incoming messages in the reception node, and also removes the extra time required to manage the copies and organize the source of incoming messages because these messages are now arriving in order. However, rendezvous protocol introduces its own

overhead. This overhead is produced by the short signalling messages that must be sent to synchronize the supply and demand of messages. The generation of these signalling messages require time in the transfer node and also in the receiver node for its processing, in the software layer. Moreover, this time required by the software level is increased by the time spent to transfer the message through the network resulting a relevant total latency. If the data packets are small, the overhead is considerable, because the creation and delivery of small signalling messages add several software instructions for a rather simple process.

4.4. Conclusions

This chapter details the implementation of the MPI standard for its use in many-soft-core systems. Two implementations have been done for shared memory architectures and another one for distributed memory architectures for systems based on Altera's NIOSII, Leon and other proprietary processors such as the ones available in STHORM platform or Recore's Xentium processor. Special emphasis has been given to present the implementation solutions both for shared memory architectures and distributed memory architectures. For that reason, it has been proved that, even if the original ocMPI library was implemented in a specific processor, its API is implementable in any soft-core processor and other embedded processors. The work done in this chapter produced a conference article [101].

The results achieved using ocMPI prove that this paradigm is a real option when programming embedded systems, and when it is compared with other alternative implementations, it is shown that all of them have similar footprint, being ocMPI the only one that is open source.

5. NoCS

Once the basic concept of NoC architectures have been discussed in previous chapters, this section covers some proposals to enhance the behaviour (and when it is possible) the performance of a NoC-based MPSoC. The general idea that rests beneath the proposals is to study software weak points, identify their functionality and, move that functionality to the NIC. These weak points are, in this work, only related to the software layers dedicated to allow or facilitate the programming of the systems. The idea is to facilitate and improve the programming of NoC-based MPSoC systems.

This is a very attractive task that could imply an almost infinite work. For that reason, this thesis has been restricted in this point to three different designs that touch three different levels of behaviour within the programmability and performance of the system: (i) the synchronisation protocol for point-to-point sharing data, (ii) synchronization primitives for the system, and (iii) data movement from memory to network.

5.1. Delivery Protocol

The mapping of an application tasks over the different cores of an MPSoC is usually critical. There is a constant risk of unbalanced workload that could penalize the overall performance and scalability of the system, since some processors could spend a huge amount of time, for example, just waiting for new work to do.

It is quite usual in MPI parallel applications that the computational burden of each process is, a priori, unknown, heterogeneous and unpredictable. In such cases, it is necessary to adopt some kind of dynamic schedule of the tasks in the processors. For

distributed memory systems, the Master-Slave model is often the chosen one. In this scenarios, the Master process is dedicated (completely or not) to distribute work and collect results, while the Slave processes are dedicated to work on its own tasks. When a Slave ends a task, it sends a signal back to the Master in order to obtain new work to do.

Therefore, a parallel system using MPI programming model with Master-Slave scheduler must introduce a global synchronization. This may be expressed as a choice between two different options: using eager protocol or using rendezvous protocol [86].

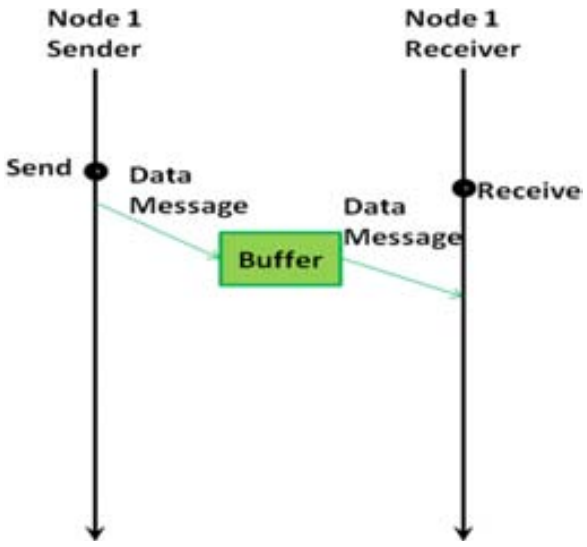


Figure 52 Eager transmission protocol.

Eager protocol, Figure 52, could achieve better results in terms of performance in some applications. However, since messages are sent regardless of the state of the receiver, incoming messages whose receive has not been posted must be stored with a buffer at the destination until a receive call is posted by the receiving MPI application. Such situation becomes worst in a multipoint communication scenario where this protocol has a great probability of going out of order at the delivery of the messages emitted from several sources. This behaviour implies the need of several buffers in the destination nodes to order the messages coming from each source.

The need of buffers in the reception nodes creates an overhead at managing the memory copies, which implies penalizations in execution time due to these copies, and in extra memory space request for the implementation of such buffers, which are a serious handicap for the lack of resources of MPSoC systems.

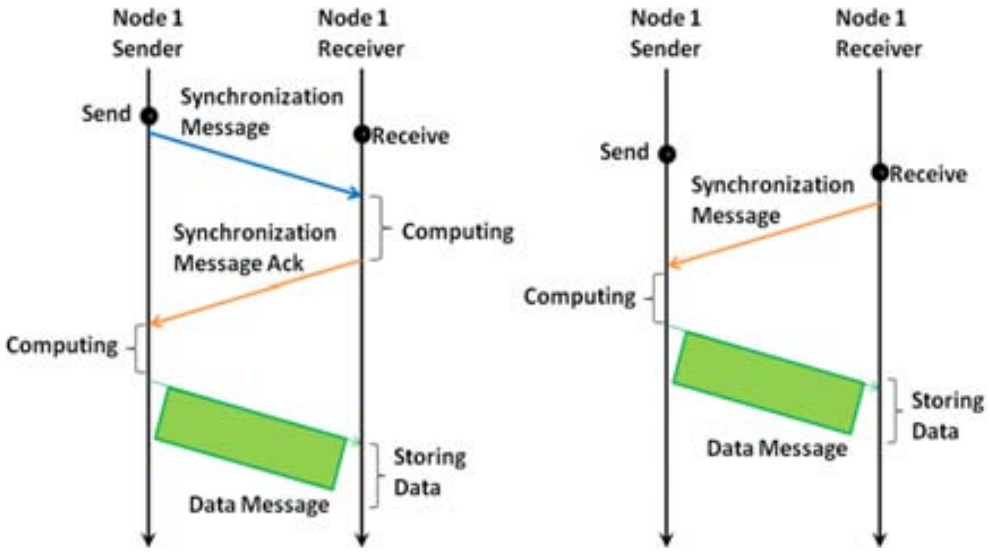


Figure 53 Rendezvous transmission protocol**. Left)** Send initiates communication. **Right)** Receive initiates communication.

In rendezvous protocol, Figure 53, there is a global flow control synchronizing all the system. The use of this protocol solves the problem of needing extra buffer space to copy incoming messages in the reception node, and also erases the extra time required to manage the copies and organize the source of incoming messages because these messages are now arriving in order. However, rendezvous protocol introduces its own overhead. This overhead is produced by the short signalling messages that must be sent to synchronize the supply and demand of messages. These signalling messages require time to be generated in the transfer node and to be processed in the receiver node, in the software layer. Moreover, this time required by the software level is increased by the time spent to transfer the message through the network, resulting a relevant total latency. If the data packets are small, the overhead is considerable, because the creation

and delivery of small signalling messages add several software instructions for a rather simple process.

In order to minimize buffering requirements, the rendezvous protocol usually implements send after receive. When send is executed before receive storage is needed to store the message in some intermediate memory causing an overhead. Moreover, in order to minimize the communication traffic, receiver node can initiates rendezvous protocol as is shown in Figure 53 right. Therefore, sender node is waiting for the synchronization message from receiver and the handshake performed by rendezvous protocol can be reduced.

Rendezvous protocol achieves better performance than eager protocol in applications where narrow cast traffic pattern is predominant, those applications using Master-Slave configuration, due to the overhead introduced by the eager protocol previously explained.

5.1.1. Offloading Delivery Protocol

In order to avoid the overhead produced by the rendezvous delivery protocol shown, we enhanced the NIC component with specific hardware that frees the processor from executing the software instructions of the protocol (Figure 53 right) in a master-slave scenario. Figure 56 shows the rendezvous process on the base line system, as it has been explained in the previous section. The Master processor generates a synchronization message, step 1 on the Figure 56. The message travels through all the layers of the system before it is injected in the NoC. Then, the slave processor receives the rendezvous message at step 2, and, finally, the data message is sent and received at steps 3 and 4. Figure 57 shows the behaviour of the proposed hardware. Once again, the

master processor wants to receive a message from the slave, but now the synchronization messages are done at NIC level avoiding spending many cycles from both processors (master and slave). Thus, the protocol becomes transparent to the programmer while reduces considerably the overhead of creation and processing of the synchronization messages.

The solution presented for this case adds the protocol to the NIC. The architecture of the NIC is divided into two modules, one focused on the interaction with the processor bus and the other focused on the interaction with the Network-on-Chip. Figure 54 shows the architecture designed for the send

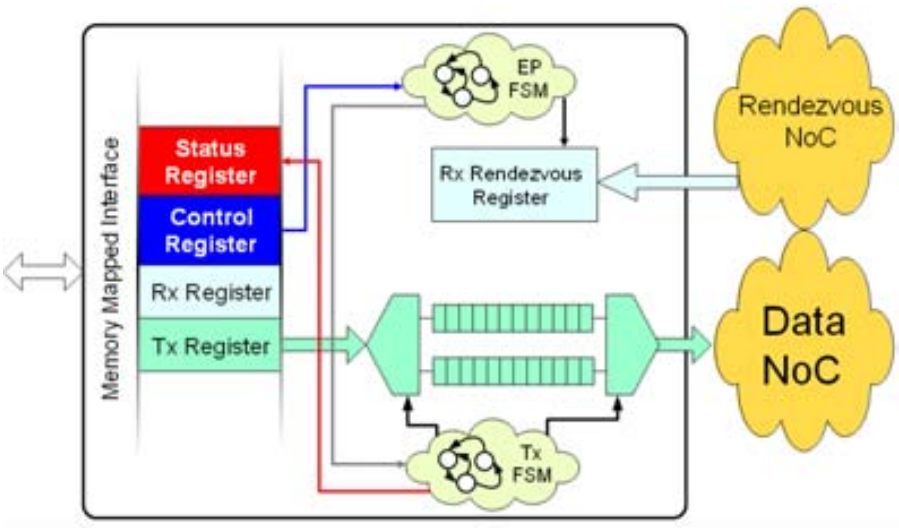


Figure 54 Network Interface Controller architecture for *send*.

Transmission of data begins when a synchronization message arrives from the Rendezvous NoC at the NIC, which indicates that the receiving module is ready to receive the message. Rendezvous NoC has been designed as a fast network on chip for small data messages, due to the switching method implemented [87]. Once the synchronization message is received, the NIC initiates the emission of the message stored on the TX register. Therefore, there is no extra software cycles consumption on the sender to analyze the synchronization message, since the protocol is now transparent. Figure 55 shows the architecture designed for receive.

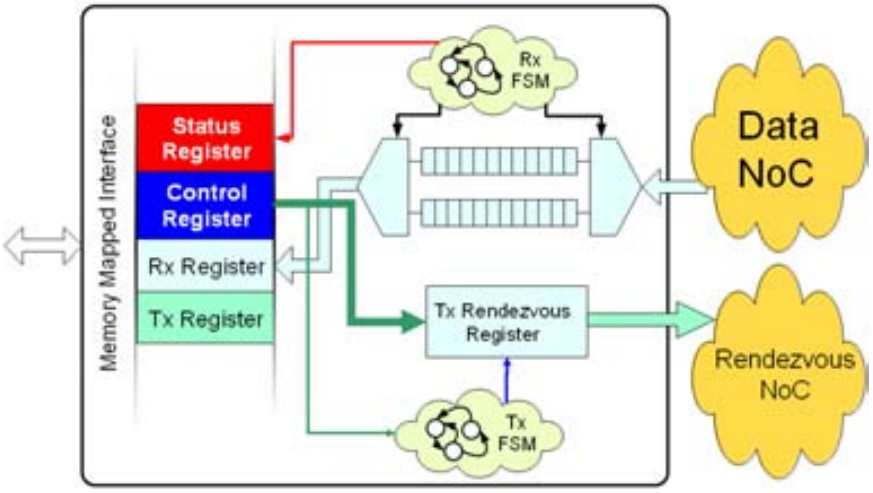


Figure 55 Network Interface Controller architecture for *receive*.

When the receive instruction is executed, the NIC is programmed with the next source to receive. Then the NIC initiates the rendezvous protocol and generates a synchronization message automatically that will be sent through Rendezvous NoC. Therefore, the receiver does not consume any extra cycles to generate the synchronization message.

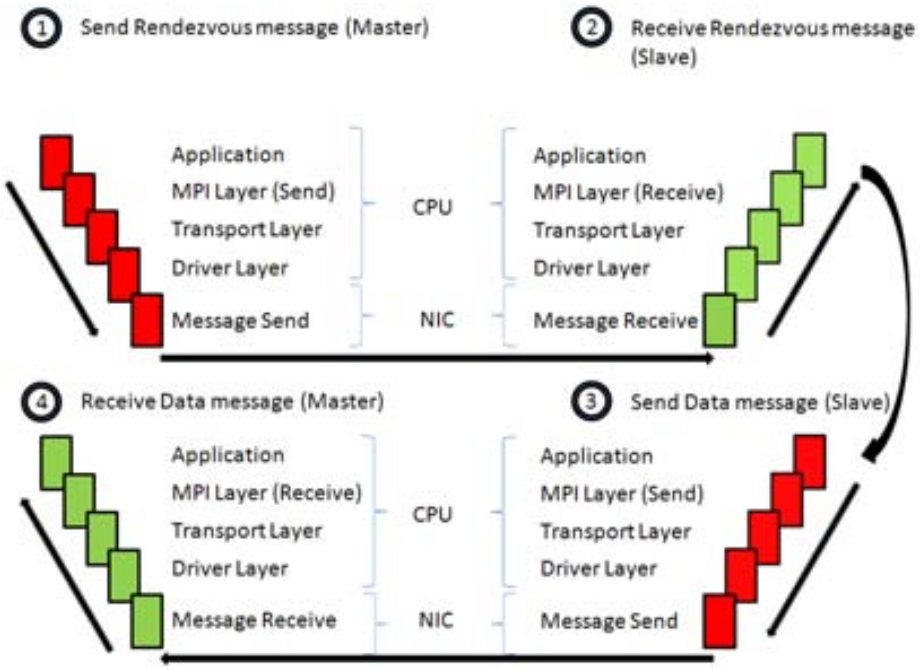


Figure 56 Software rendezvous scheme over NoC-based MPSoC.

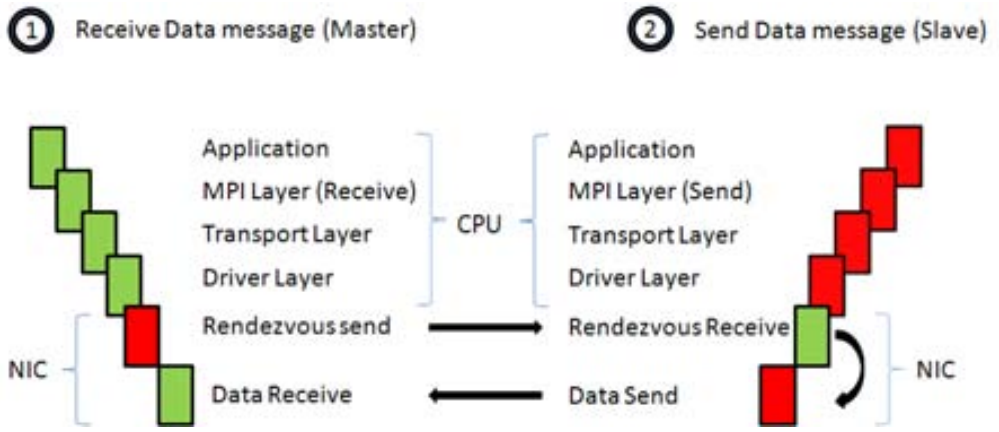


Figure 57 Hardware rendezvous scheme over NoC-based MPSoC.

The rendezvous network is based in an ephimeral network [87]. This network is based in a point to point single flit connection. It is a low latency network when used to transmit one single flit of information. This behaviour matches perfectly with the requirements of the rendezvous network.

All the modules designed in this paper including NIC and NoC, have been designed and generated by NoCMaker tool.

5.1.2. Implementation and Results

The experiments for this system have been performed using a Stratix II EP2S180 DSP development board. The baseline system is a NoC-based system with 16 NIOSII soft-core interconnected with a 4x4 2D mesh, as it has been explained in section 3. Table 4 show the resource occupation of the interface and rendezvous network designed. The 16 NIOSII system with Rendezvous solution occupies 60207 Combinational ALUTS, which represents an overhead of 2 % with respect to the same system not using this solution. Analyzing the assembly code of the MPI protocol, the solution proposed reduces up to 596 and 550 instructions for send and receive packets respectively. These instructions are the amount of instructions of all the software layers that are in charge of generate the rendezvous messages. Consequently, execution time is reduced up to 2122

and 1995 cycles for every send and receive packet in the system when counting all layers.

	Combinational ALUT	Dedicated logic registers
NIC	237	1060
NIC + Rendezvous protocol	242	1070
Wormhole Network (without NIC)	1063	769
Rendezvous Network (without NIC)	5803	5728

Table 4 FPGA Synthesis Results NIC and NoC

We tested the design with an application to compute the Mandelbrot set for an 800 x 600 image, by using a master-slave work-sharing pattern. In this particular implementation, the master keeps distributing every pixel to the slave nodes and collecting the computed results until all the pixels of the image have been computed.

This approach becomes very challenging because as we fragment the data that is distributed to slaves at pixel level, the overhead of communication is very high.

On the other hand, the ratio between communication and computation is quite high for a large number of the pixels that are outside the Mandelbrot set. In a typical supercomputer, such application would show no speedup at all due to these issues.

In our MPSoC platform we still get some speedup (as shown in Figure 58), but being below factor 2x what does not justify the use of a multiprocessor.

When using the hardware assisted design, scalability profile is much better, being able to reach a speedup factor of 6x for 12 processors. Within an ideal system in which communication costs were negligible, we could expect a maximum speedup factor of

11x for 12 processors, because the master node is only dispatching the workloads among the 11 worker slaves and not doing any computation.

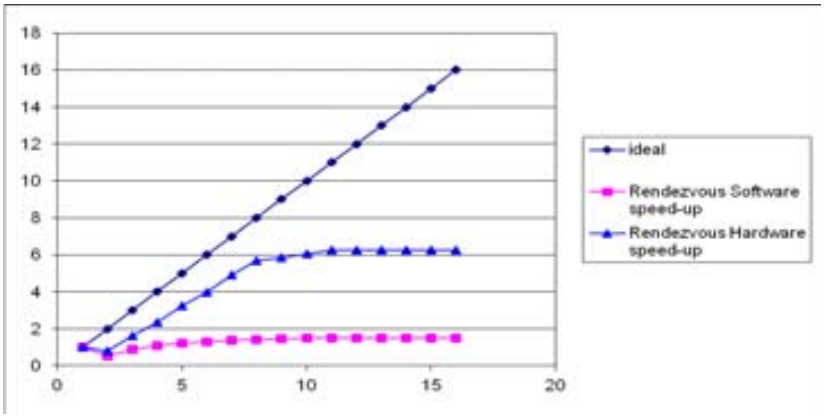


Figure 58 Scalability of Mandelbrot Set application for the original ocMPI implementation (pink) and for the version with Hardware Assisted Rendevouz.

5.1.3. Summary

In this implementation, it has been presented an effective design to reduce part of the overhead that is generally associated with message passing programming models. A hardware implementation of the rendezvous protocol presents several benefits; first it minimizes the need of transmission buffers, which is a critical factor in embedded systems (in FPGAs), while avoiding the execution of the associated software. The off-loading of the process to the Network Interface Controller allows a more performant MPI applications while makes easier, to the application programmer, the global system synchronization. Moreover, the solution presented does not have a significant impact on the resource occupation of the network interface. However, the design will not be of a great impact on performance when the application uses a communication pattern with low level of messages and great amount of data.

5.2. Bus Master

Nowadays, one of the bottlenecks for the improvement in performance is the existing gap between the CPU speed and the memory speed. For the last decades the CPU speed improved at annual rate of 55%, however the memory access speed improved only at 10%..

Patterson et at. in [97] present the named Three walls. These three impediments defined the end times of increased computing performance. They would prevent computer users from ever reaching, for instance 10 GHz Pentiums. These three walls are known as:

"Power Wall + Memory Wall + ILP Wall = Brick Wall"

- The Power Wall means faster computers get really hot.
- The Memory Wall means 1000 pins on a CPU package is way too many.
- ILP[6] Wall means a deeper instruction pipeline really means digging a deeper power hole.

Taken together, they mean that computers will stop getting faster. Furthermore, if an engineer optimizes one wall he aggravates the other two.

Figure 59 shows the performance of a mono-processor versus the performance improvement in time to access the main memory. With the processor line, Dr. Patterson wants to show the memory requests per second (measured on average), while with the memory line he wants to show the provided DRAM accesses per second.

For example, the Intel core i7 processor can generate two data memory references per core each clock cycle; with four cores and a 3.2 Ghz clock rate, i7 can generate a peak of 25.6 billion 64-bit data memory references per second, in addition to a peak

[6] ILP stands for instruction level parallelism

instruction demand of about 12.8 billion 128-bit instructions references, this is a total peak bandwidth of 409.6 GB/Sec. In contrast, the peak bandwith to DRAM main memory is only of 25GB/sec (only 6% of i7) [88].

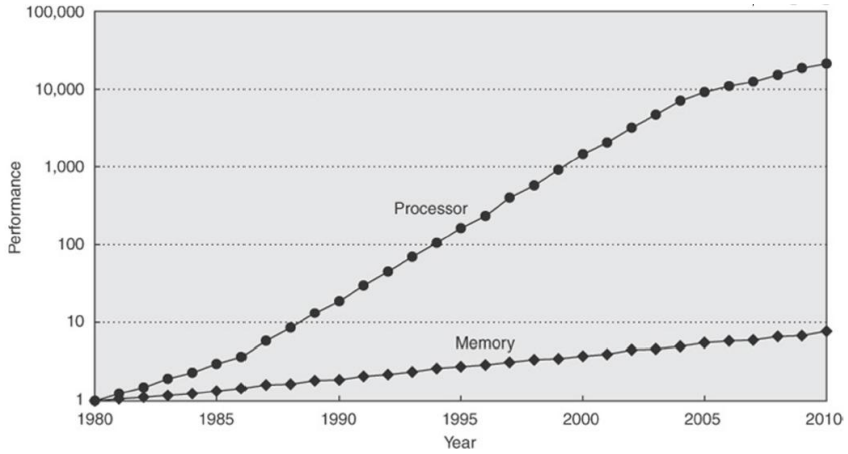


Figure 59 Performance evolution through time processor versus memory.

A possible solution to this problem would go through to off-loading the most used memory-related instructions, these that have relation with the transferring data between several memory units, or two distinct memory addresses. This is the memory copy instructions.

The *memcpy*, *memmove* and so on are intensively used in message passing implementations such as MPI, and also are widely used in operating system routines, device drivers and in network managing.

In *memmove*, copies the values of *num* bytes from the location pointed by source to the memory block pointed by destination. Copying takes place as if an intermediate buffer were used, allowing the destination and source to overlap.

The underlying type of objects pointed by both the source and destination pointers is irrelevant for this function; the result is a binary copy of the data. The function does not check for any terminating null character in source - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed by both source and destination parameters, shall be at least *num* bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach). *memcpy*, copies the values of *num* bytes from the location pointed by source directly to the memory block pointed by destination.

These operations are usually performed by the main CPU. That means that the processor is stalled doing memory transfers from the main memory to an inner memory in the processor, and then to the target peripheral, as for example the NIC and from there to any other point in the network, usually another memory.

Optimizations to memory copy related functions, which are always the most time-consuming parts of many programs, have been proved very effective in promoting performance of system or I/O device [89].

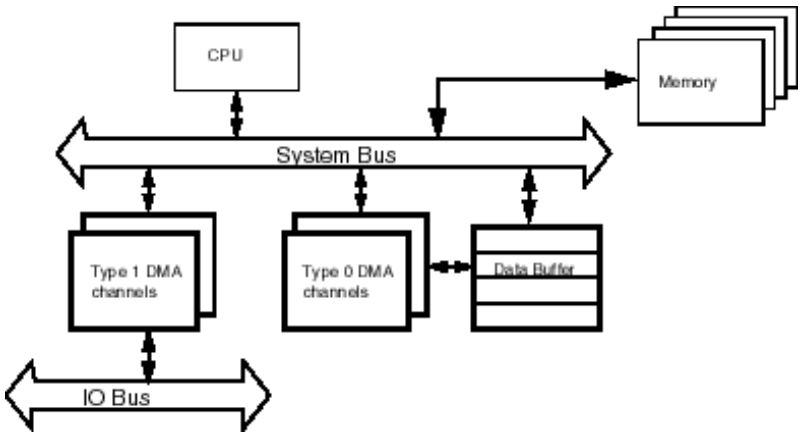


Figure 60. Generic system with a DMA controller diagram block.

The most common hardware solution is related to the use a co-processor attached to the system bus, Figure 60, which can free the main CPU from moving the data between source and destination. This co-processor is known as DMA (Direct Memory Access) controller or DMAC (many hardware systems use DMA including disk drive controllers, graphics cards, network cards and sound cards).

Therefore, a DMA controller is a peripheral focused on perform data transfers on behalf of the CPU. The usual flow of using a DMA is:

1. DMA is set up from CPU. Some DMA registers are programmed to enable DMA transfers. This is a combination of software and hardware operations. Example shown in Figure 61

```
perform_dma_operation(oc_id, source_address, dest_address, callback_ID, count)
{
        // the software needs to get the put_pointer
        Software_put_pointer->status = 0xFFFFFFFF // DMA engines request init value
        Software_put_pointer->source_address = source_address;
        Software_put_pointer->destination = dest_address;
        Software_put_pointer->count = count;
        Software_put_pointer->callback_function_ID = callback_ID;
        Software_put_pointer->operation_control_id = oc_id; //reference to the initiator of the DMA
        Flush(software_put_pointer) // optional if data in L1 cacheable
        // start DMA engine if it is not running.
        Software_put_pointer++; //check for wrap and other conditions
}
```

Figure 61 Code example for perform_dma_operation function.

2. DMA starts copying data from the source memory to an internal buffer. To buffer temporally the source data allows timing decoupling between the two main actors devices, these are the source memory and the target device. Additionally, the internal DMA buffers simplify transfer between devices with different bursts.
3. Data is finally moved to target destination device.
4. CPU checks the status of the operation to know if transfer is done or not. This control operation can be done by pulling or by interrupt (Figure 62)

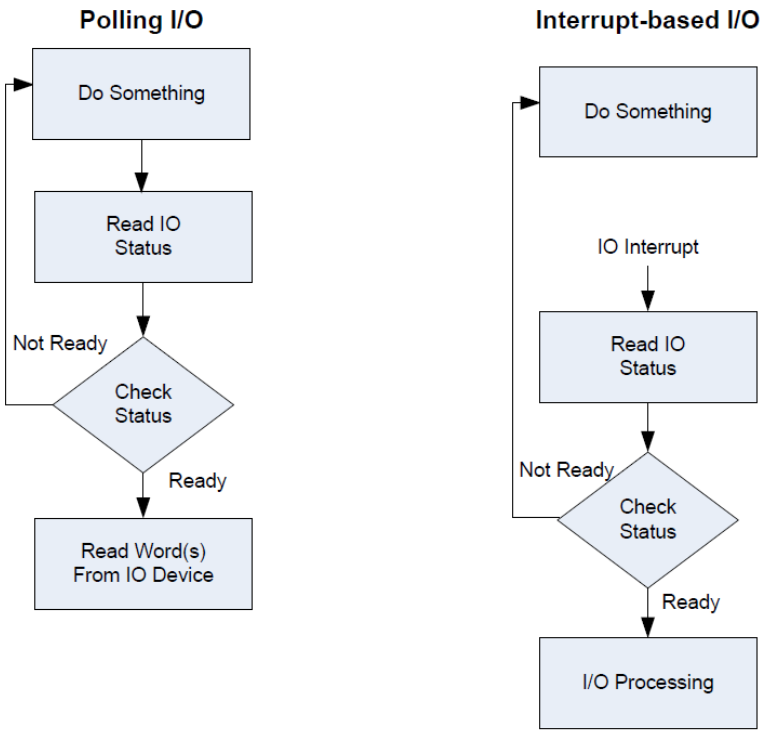


Figure 62 DMA controller notification mechanisms.

Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel. Similarly a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time and allowing computation and data transfer concurrency.

Without DMA, using programmed input/output (PIO) mode for communication with peripheral devices, or load/store instructions in the case of multicore chips, the CPU is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done. This is especially useful in real-time computing applications where not stalling behind concurrent operations is critical. Another and related application area is various forms of stream

processing where it is essential to have data processing and transfer in parallel, in order to achieve sufficient throughput.

DMA is also used for intra-chip data transfer in multi-core processors, especially in multiprocessor system-on-chips, where its processing element is equipped with a local memory (often called scratchpad memory) and DMA is used for transferring data between the local memory and the main memory.

In this work, we present an alternative to off-load the memory move related instructions from CPU to the network interface controller.

In this work, we proposed to completely remove the DMA from the multi-core system. To do so, we want to off-load the memory transfers control to the network interface controller. The flow is similar to the DMA flow:

1. TRANSFER PROGRAM: CPU programs the transfer of data by setting up some NIC registers. When on the application is required to send data, MPI_Send is called. Therefore, MPI layer, which relays over several layers, flows to the immediately lower layer which is the driver layer. The low-level driver operations are in charge to program the NIC control register.
2. COPY DATA: NIC starts buffering data from source data using its the master bus interface, while the CPU can continue processing code. Additionally, NIC also transfers the data through the network to the destination, where the target NIC stores de incoming data to the address that CPU has programmed when MPI_Recv is performed.
3. MOVE DATA: Data is sent through the network.
4. NOTIFICATION: CPU checks the status register of the NIC to know when the transfer is done. The notification phase can be implemented, also, as an interrupt mechanism.

5.2.1. Implementation and Results

Figure 63 shows the breakdown of the time consumed by the MPI_SEND primitive of the UAB MPI library for distributed systems, and the code used to implement the low level function sendTo (driver layer).

Figure 64 shows that the CPU is wasting great amount of time preparing and writing the data flits to be sent. So, in an intuitive way, it seems obvious that offloading that part to the NIC will result in an increase the performance of the system since the CPU is free to perform other operations. However, it could take place that the CPU, once freed from these tasks, has not any other task to perform, but this is the programmer duty to avoid that situation. Additionally, the offloading design introduces its own overhead.

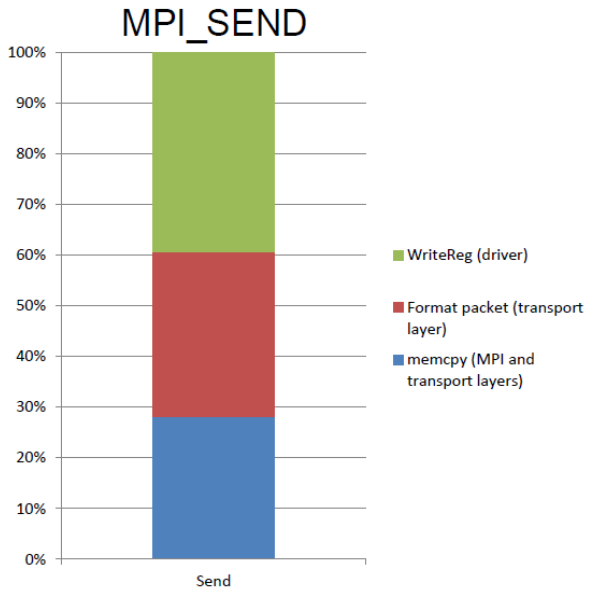


Figure 63 Breakdown of the MPI_SEND primitive (sending 1024 bytes of data).

As mentioned before, the study and design of this functionality has been based on the ocMPI implementation. Despite the fact that this is a specific design done for a specific MPI library implementation, the general idea can be abstracted and bring to any other MPI implementation.

```
/**
 * Sends 'length' bytes from buffer to destination address
 * We pack up to two data bytes in eack packet.
 * @param buffer Data payload to be sent (represented using an array of bytes)
 * @param length Number of bytes to write
 * @param address Destination address
 */
void sendTo(int x, int y, byte *buffer, int length)
{
        ...
          do {
                nicWhReadReg(NWH_STATUS_REG, &status);
          } while (status & NWH_STATUS_TX_BUSY);
        /**
        * First flit of each packet carries the addressing bits and up to
        * two bytes of payload
        */
        #ifdef NIOSII
                flit = (PACKXY_IN8(x,y) << 24) |
                PACKXY_IN8(INDEX_TO_ADDRESS_X(NIOS2_CPU_ID_VALUE),
                INDEX_TO_ADDRESS_Y(NIOS2_CPU_ID_VALUE)) << 16) | (buffer[0] << 8)  |
                (buffer[1]);
        #endif
        nicWhWriteReg(NWH_TX_REG, flit);  // write first flit to TX register
        nFlits++;
        /*
        * Other flits carry 4 bytes of payload
        */
        for (n=0; n < nFourByteFlits; n++, j+=4, nFlits++)
        {
                flit = (buffer[j] << 24) + (buffer[j+1] << 16) + (buffer[j+2] << 8) + buffer[j+3];
                nicWhWriteReg(NWH_TX_REG, flit);
        }
        nicWhWriteReg(NWH_CONTROL_REG, NWH_CONTROL_TX_START_FSM);
        /*
        * Other flits carry 4 bytes of payload
        */
        for (n=0; n < nFourByteFlits; n++, j+=4, nFlits++)
        {
                flit = (buffer[j] << 24) + (buffer[j+1] << 16) + (buffer[j+2] << 8) + buffer[j+3];
                nicWhWriteReg(NWH_TX_REG, flit);
        }
        nicWhWriteReg(NWH_CONTROL_REG, NWH_CONTROL_TX_START_FSM);
}
```

Figure 64 Low level function from driver to perform a send for a Wormhole NoC.

The new NIC design is divided in the following modules (**Figure 65**):

- Registers bank.
- Buffers.
- System bus interface (or Bus control module).
- Inner NoC-Router interface.

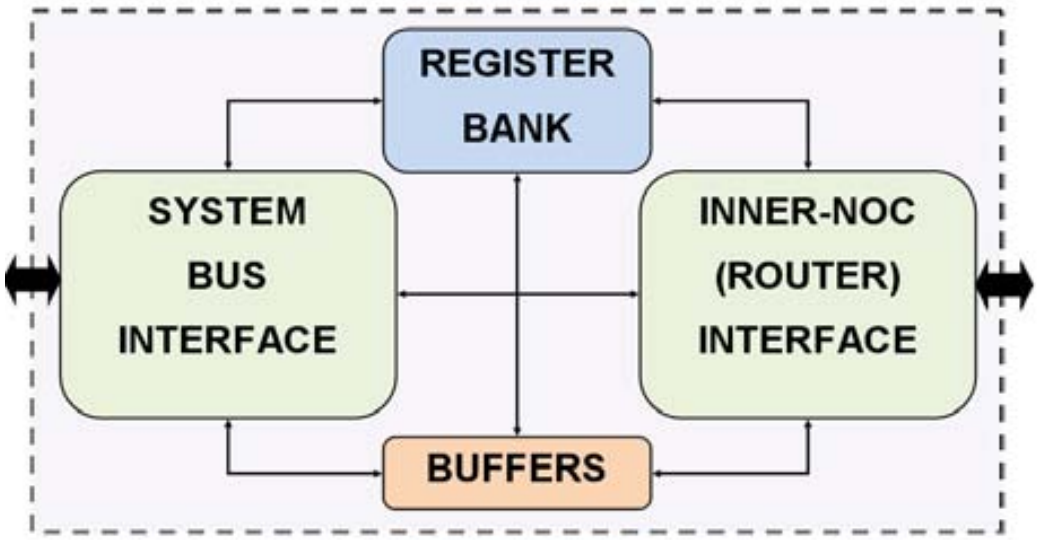


Figure 65 Block diagram of the Bus Master NIC**.**

- ## Registers Bank

```
/*
*    BUS ADDRESS              SELECT
*       0                  DATA SIZE TO RECEIVE
*       1                  DATA SIZE TO SEND
*       2                  ADDRESS TX
*       3                  ADDRESS RX
*       4                  IS TX
*       5                  IS STATUS
*       6                  IS CONTROL
*       7                  IS RX
* */
```

There are 8 x 32bits registers on the NIC that can be accessed from the bus to control the network interface controller.

The first four registers (RxDataSize, Tx dataSize, Tx Address, and RxAddress) are used to control the NIC section that controls the transfers with the processor bus.

1. RX DATA SIZE REGISTER. This is a 32 bit register that stores the number of data words to receive by the posted receive instruction.

   Offset 0x0 RxDataSizeRegister

   This register is used internally by the NIC to ensure that the correct amount of data is received through the network and, stored in the final target memory.

2. TX DATA SIZE REGISTER. This is a 32 bit register that stores the number of data words to transfer by the posted send instruction.

Offset 0x04 RxDataSizeRegister

This register is used internally by the NIC to ensure that the correct amount of data is moved from the source memory and, send through the network.

3. TX ADDRESS REGISTER. This 32 bit register is used to indicate the address where the data, to be transferred, is stored.

   Offset 0x04 TxAddress

   This register is used internally by the NIC as the initial offset address to access the memory where data is stored.

4. RX ADDRESS REGISTER. This 32 bit register is used to indicate the address where the incoming data from the network must be stored.

   Offset 0x08 RxAddress

   This register is used internally by the NIC as the initial offset address to store the data incoming from the network.

The remaining registers are used to control the entire NIC.

1. TX REGISTER. This is a 32 bit register that can be used by the CPU to directly transmit data to the NIC. The CPU can use that register to transfer data instead of using the DMA-like functionality of the NIC.
2. RX REGISTER. This is a 32 bit register that can be used by the CPU to directly read data from the NIC. The CPU can use that register to get data instead of using the DMA-like functionality of the NIC.
3. STATUS REGISTER. 32 bit register that informs about the status of the network controller.
   - Bit 0: done
   - Bit 1: eFIFORx
   - Bit: DA

- Bit RxUsedw

4. CONTROL REGISTER.

System Bus Interface

This module is in charge of the data transfers from the target system memory to inner buffers of the NIC, and also, it is in charge of the communication with the CPU. Within this module there are two main components. The first one that manages the protocol of the system bus, and another one that manages the amount of data moved. Figure 66 and Figure 67 show the diagram block of the components inside the module.

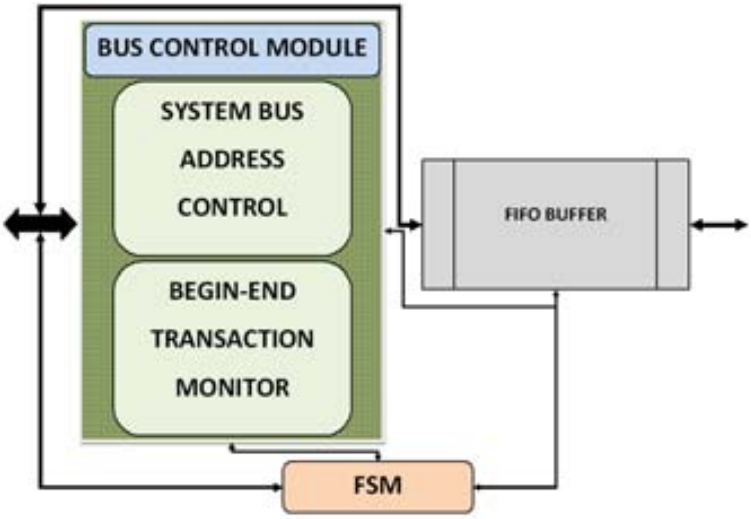


Figure 66 Block diagram of the bus control module**.**

In the implementation example implemented within this work, the bus interface is the Altera's Avalon bus [65]. The reason is the physical availability of design kits from Altera in the Lab. So therefore, we can do a real implementation over on FPGA of the proposal presented in the thesis, instead of just simulate it.

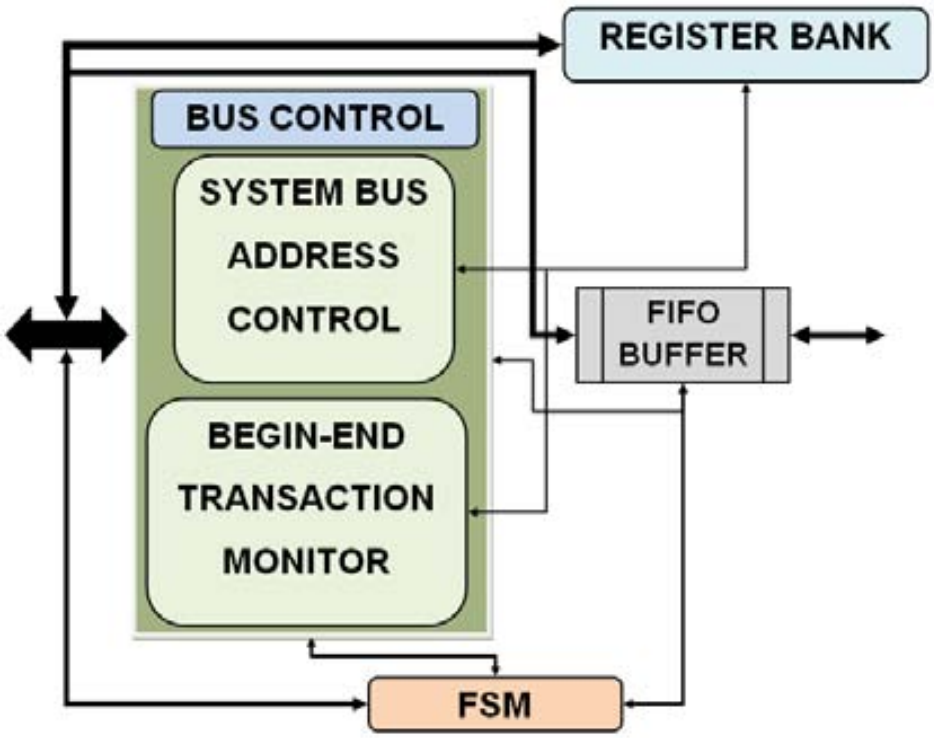


Figure 67 Block diagram of the bus control module interacting with the register banc and NIC buffer**.**

The core of the system bus address control module is a counter register that updates the value of the address to drive into the master Avalon bus interface. This sub-module requires 31 combinational ALUTs and 32 registers, Figure 68 shows the RTL view extract from the synthesis process of QuartsII tool. .

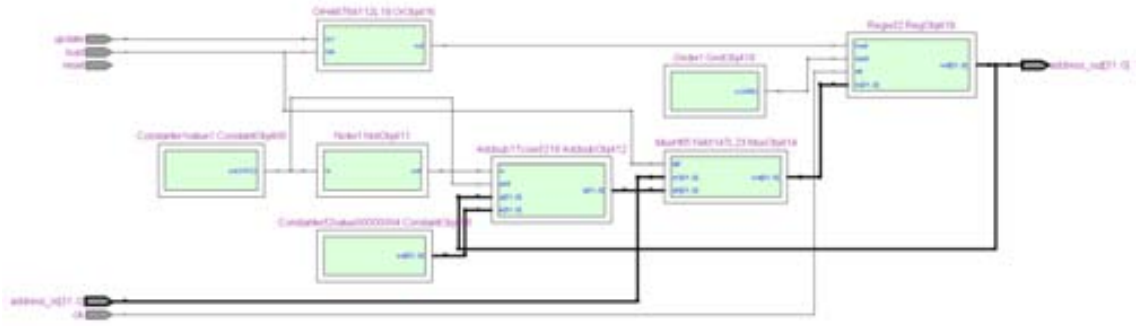


Figure 68 Address control module RTL diagram from QUARTUSII**.**

The main task of the second part of the system bus interface, the so called begin-end transaction monitor, is to control the amount of data transferred from – or to – the

memory. This sub-module requires 75 combinational ALUts and 32 registers; Figure 69 shows the RTL view extract from the synthesis process of QuartsII tool.

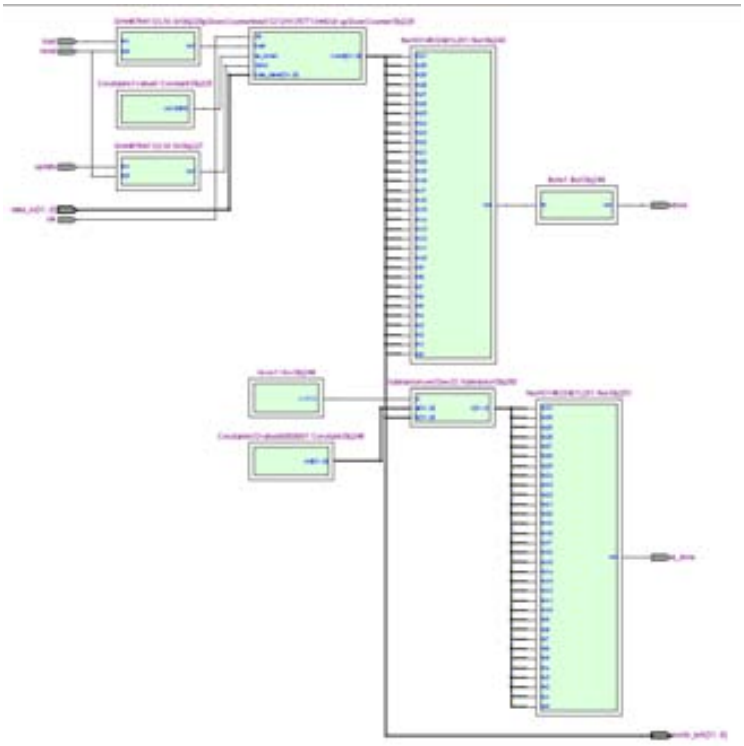


Figure 69 Transaction monitor module RTL diagram from QUARTUSII**.**

The experiments of this work use a Stratix II EP2S180 DSP development board and a Cyclone III Edition NIOSII embedded Evaluation Kit. In summary, the system bus interface module (or bus control module) requires around 44 combinational ALUTs and 67 registers (using Quartus II and Cyclone II EP3C25F324C6).

- ## Inner-NoC-router interface

Within this module there are implemented the configuration of the NOC system, which are the routing protocol and, switching method. In this module, the data is packetised into wormhole flits and send to the router in 32 bits flits using a four-phase handshake.

- ## Results

Inside the NIC there are two buffer systems, one to store data incoming from the system bus and driven to the router, and the other one in the other way around.

In summary, the new design uses 275 ALUTs and 702 dedicated registers (Table 5).

	Combinational ALUT	**Dedicated logic registers**
NIC	237	1060
NIC + Bus Master	275	702

Table 5 Comparison between the Bus Master NIC design versus the base-line NIC.

Figure 70 shows the throughput achieved by the design when performing data transfers of distinct sizes, and when performing matrix multiplication operation.

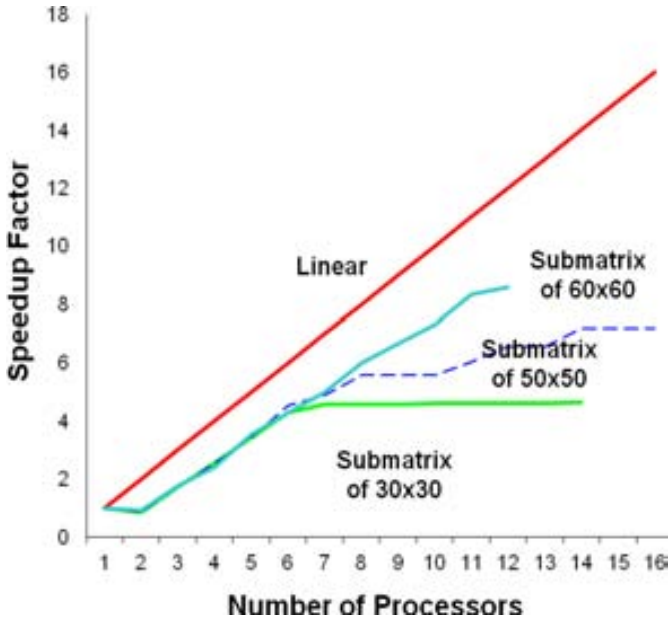


Figure 70 Matrix multiplication speedup results**.**

Additionally, Figure 71 shows the performance when an application using a high ratio of communication and computation is used like Mandelbrot set.

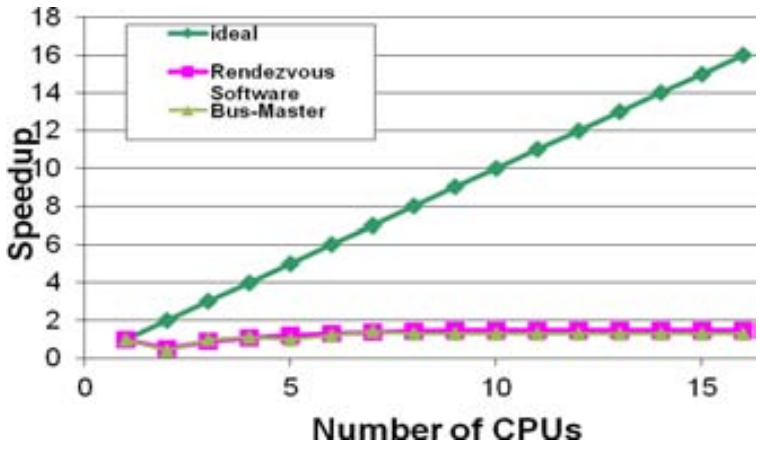


Figure 71 Mandelbrot set computation speedup results for Bus-Master NIC**.**

It can be seen from the tests that the higher is the amount of data to transfer, the better is the result obtained when using the new facility, achieving a speedup factor of 5, 7, and 8 for matrix multiplication tests, while previously we achieved no more than 2, 5 and 4 (**Figure 27**). Again, these results are limited by the on-chip memory of the small FPGA device targeted. The results would be better using devices with more internal memory.

Additionally, It must be said that when performing the Mandelbrot set computation result in almost no speedup. The results obtained are practically identical to the software-based ones. It must be said that this computation are done in a very fine grain sending to each computing point a single pixel to compute. The reason of this result is explained by the overhead when programming the NIC.

5.2.2. Summary

In this part, it has been presented an initial design to avoid the overhead produced when moving data from memory to NIC when a MPI_SEND or MPI_RECV are called.

The throughput achieved the design is far better than the one achieved when it is not used and the transfers are done by the CPU. This design presents some benefits; frees

the main CPU of spent cycles on transfers from the memory to the NIC, while increases the performance of the same transfers since it is no longer necessary to move data from memory to CPU and then again from CPU to NIC. With this design the data is moved from memory directly to the NIC.

However, the performance of the design could be even better when implementing burst processes. Despite of that, the design imposes some overhead since, the bus-master NIC must be programmed by the processor, which is translated in some bus accesses to the NIC. Even more, when the amount of data to transfer is high it becomes necessary to re-program the network interface controller.

5.3. Esyncop

For enhanced synchronization operations. In MPI applications is usual to find different synchronization operations. This optimization will reduce the time required for a single node (typically the master of the system) to send the same information to the rest of the system. This design will improve INIT, FINALIZE, and BARRIER MPI functions.

There are six MPI basic functions required to implement any application using MPI programming model. However, MPI_Init and MPI_Finalize are completely essential for an application based on MPI library [90]. MPI_Init sets up the MPI environment. The MPI standard does not specify what to do before MPI_Init, however before MPI_Init no MPI call will work. Therefore, before MPI_Init it should be done as little as possible, moreover it should be avoided any code that could change the external state of the program (for instance, I/O operations).

In general, distinct implementations of MPI have different MPI_Init work. However, the duty of this primitive is to create, initialize, and make available the MPI

environment. Since the MPI standard does not specify HOW, each MPI implementation writes the MPI_INIT in different ways.

Therefore, any MPI code should be like shown in Figure 72:

```
#include “mpi.h”
#include <stdio.h>

Int main(int argc, char **argv)
{
        MPI_Init(&argc,&argv);
        Printf(“Hello world”);
        MPI_Finalize();
        Return(0);
}
```

Figure 72 Simple “Hello world” MPI code example**.**

When talking on embedded MPSoCs, or MPSoCs over FPGA, as in this work, it is usual to find no operating system working on the distinct processors where MPI library can rely.

This is for example the case of this work, where we are working with ocMPI library that can be used without any operating system or daemon, due to the scarcest of memory resources available in such kind of systems.

Therefore, the MPI_Init implementation of the ocMPI library must do the following operations:

1. State to each processor of the system involved in the communicator a unique ID identification.
2. Inform to each processor in the communicator the full size of the system, that is, the total number of processors involved in the communication.

To do so, the so called master processor is in charge to send all these information to each and every processor, which means to send at least two integers from the master processor to the rest of slave processors. The code in Figure 73 shows the implementation of the ocMPI_Init primitive.

```
int ocMPI_Init(int *argc, char ***argv)
{
        int i=0;
        PRNT("Starting ocMPI_Init()...\n");
        /* Ensure that we were not already initialized or finalized */
        if(ocMPI_finalized) return ocMPI_ERR_OTHER;
        else if(ocMPI_initialized) return ocMPI_ERR_OTHER;

        strcpy(processor_name, ALT_CPU_NAME);

        //Initialize the global ocMPI_Global_rank variable of each process/core
        //Initialize the global ocMPI_Global_size
        if(root)
        {
            for(i=1;i<nCPUs;i++)
                {
                Transport_sendToSync(INDEX_TO_ADDRESS_X(i), INDEX_TO_ADDRESS_Y(i), &i,
4);
            }
            for(i=1;i<nCPUs;i++)
            {
                Transport_sendToSync(INDEX_TO_ADDRESS_X(i), INDEX_TO_ADDRESS_Y(i),
&nCPUs, 4);
            }

            ocMPI_Global_rank = 0;
            ocMPI_Global_size = nCPUs;

        }
        else
        {
            PRNT("ocMPI_Init: Slave node\n");
            ocMPI_Global_rank = driverRecvInt2(-1,-1);
            ocMPI_Global_size = driverRecvInt2(-1,-1);

        }
        ocMPI_initialized = TRUE;
        return ocMPI_SUCCESS;
}
```

Figure 73 ocMPI_Init code.

5.3.1. Implementation and Results

Figure 74 shows the time required by the ocMPI to initialize each node of the system. As it can be seen, the average time required per node is around 1000 cycles. Even this numbers does not seem to be a great evil, the design will be on great benefit since, not only improves the performance of the primitive, but to free the processors of such task.

The main processor can use these detached clock cycles to prepare the rest of the application, while MPI_Init is been performed by the NIC. Furthermore, some modules implemented in this design can be used in future works to off-load more primitives.

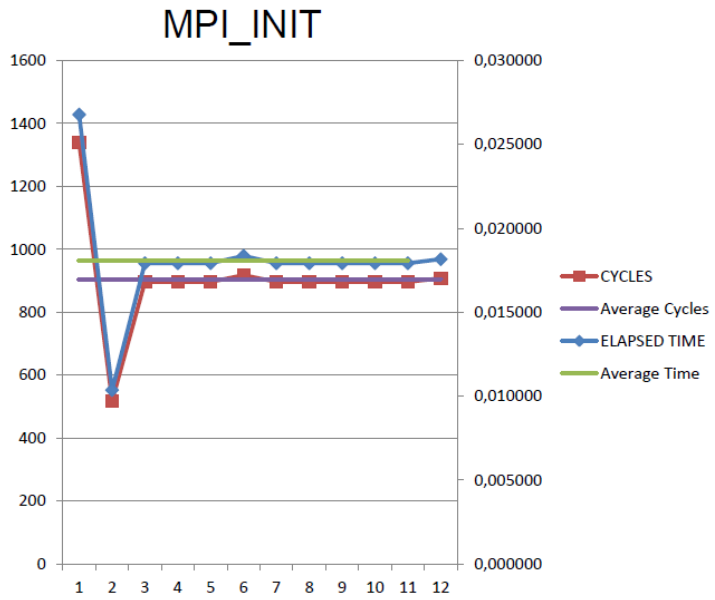


Figure 74 Time required performing MPI_Init per processor node at 50 MHz**.**

The design of this NIC enhancing facility has been implemented over Cyclone III Edition NIOSII embedded Evaluation Kit. The results showing that the Esyncop NIC design requires about 364 ALUTs and 1101 dedicated registers Table 6. Following, we are going to explain how these resources are distributed.

	Combinational ALUT	**Dedicated logic registers**
NIC	237	1060
Esyncop NIC	364	1101

Table 6 Comparison between the Esyncop NIC design versus the base-line NIC.

Figure 75 shows the generic diagram bloc of the NIC design.

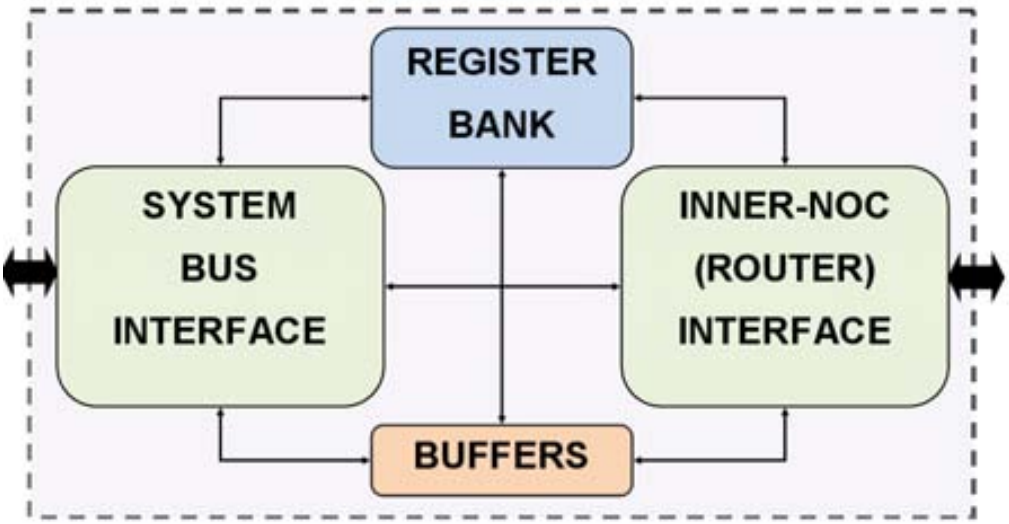


Figure 75 Generic NIC block diagram.

The effort in the design of this case was focused on the Inner-NOC interface module, Figure 76.

This module is divided in three sub-modules, as it can be seen in the 0. It must be said, that the special sub-module that enables the searched behaviour is the so called synchronization unit module. This module implements de generation of the data flits required for the MPI primitives INIT/FINALIZE.

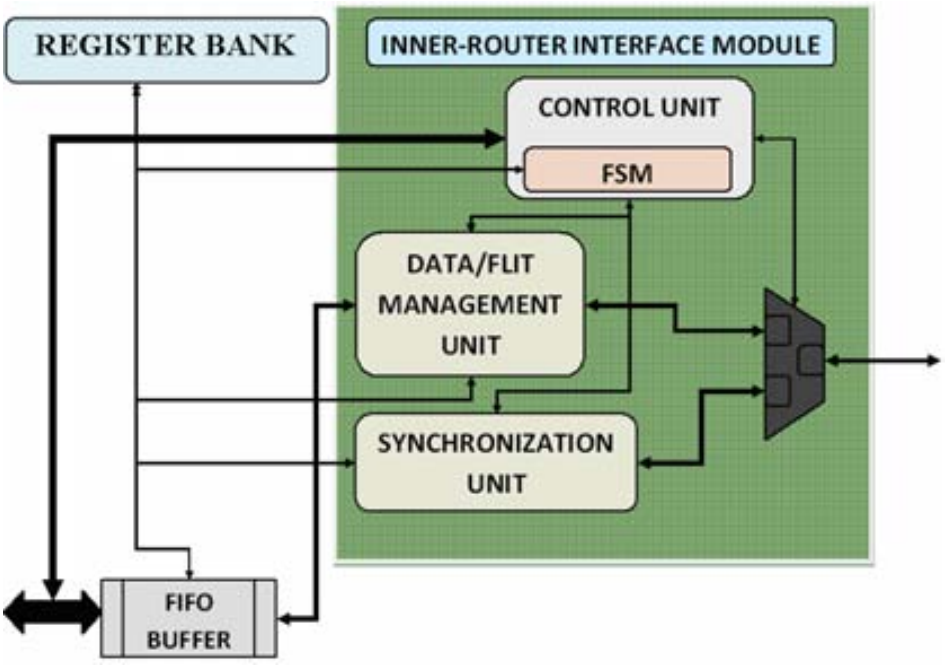


Figure 76 Inner-router interface module block diagram**.**

Within the synchronization unit is implemented the generation of a unique identification value used in the MPI library. This value must be unique within the MPI communication world, and this is achieved using a low level counter register, and it is generated only in the master node.

Additionally, the synchronization unit has implemented an address generation that is used to address the auto-generated packets. The address must be generated following the specification of the routing algorithm used in the NOC. For the design implemented in this example, the algorithm used is a simple XY [61], Figure 77.

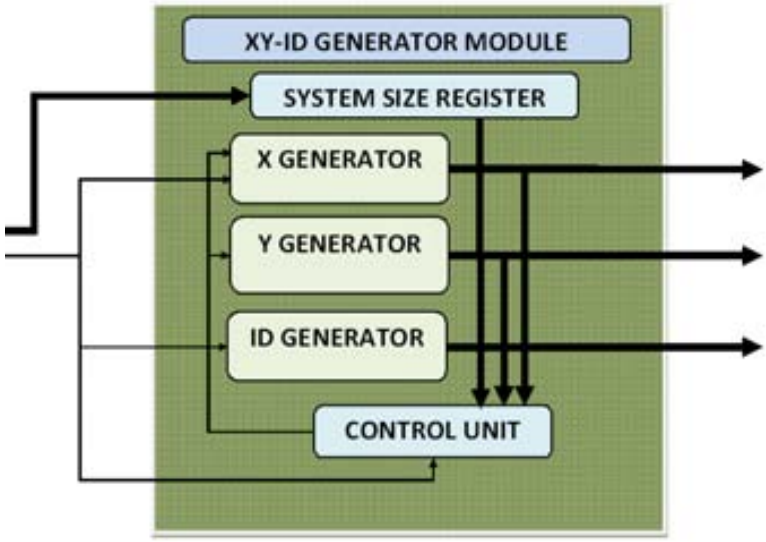


Figure 77 Address(XY)-ID module block diagram**.**

Figure 78 shows the RTL view of the Address-ID module extract from QuartusII.

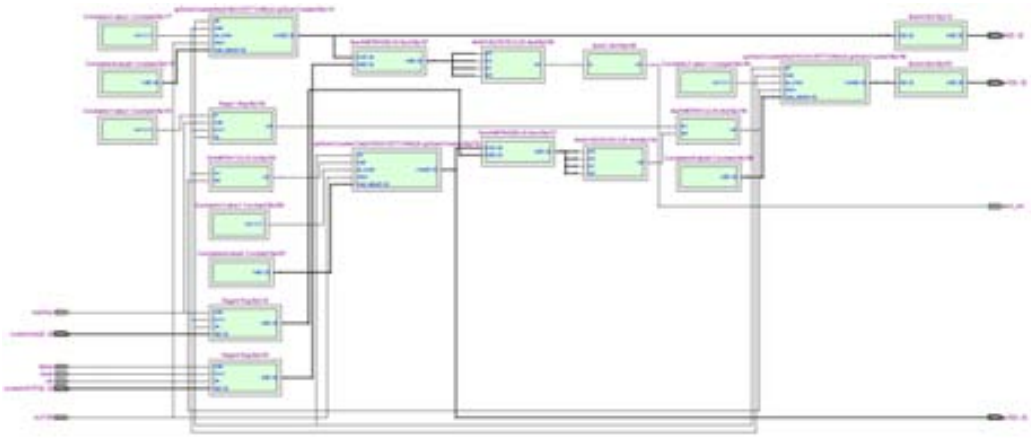


Figure 78 Address(XY)-ID module RTL extract from QuartusII**.**

Finally, when this design is used, the system performs the initialization of a 4x4 regular mesh NoC-based MPSoC in about 1000 cycles. Figure 79 shows the execution time in clock cycles consumed to initialise several NoC-based MPSoCs with a 2D-mesh topology. In the comparison, it can be seen that the original execution without any optimisations uses around 1k cycles per node to be initialised (as shown at the beginning of the section), while the optimisation software-based system improves enormously the initialisation of the MPSoCs. The optimisations consists in using low level function calls to the NIC driver, instead of using the transport layer calls as shown in Figure 73. Finally, both software-based implementations are compared with the Esync NIC design. The design shows a huge improvement over both software implementations. Even more, software based implementations trend to constantly increase the initialisation time while increasing the size of the system, however, the hardware based systems does not increase barely.

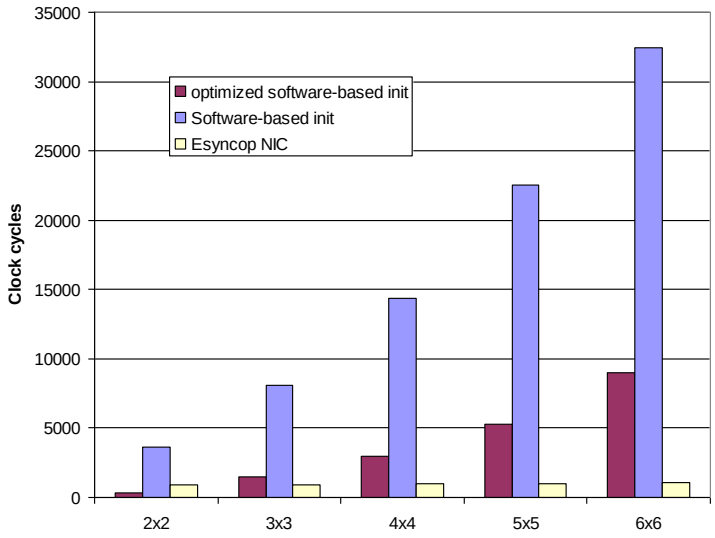


Figure 79 MPI_init performance comparison results**.**

5.3.2. Summary

This implementation presents the off-loading of the initialisation process from the processor to the NIC.

Even if the improvement achieved in the *init* process is not of a great relevance, the design contributes to release the processors of performing such task, and therefore, allowing to spend such cycles to prepare any other task. It must be highlighted, also, that the impact on resource utilisation of the design is minimal compared with the base system NIC.

Additionally, the design can be re-used in a future work to off-load other MPI primitives, which have more performance impact overall system, like broadcast primitives. Specifically, the module that generates automatically the packet address should be the same used to generate the address on a hypothetic broadcast NIC design.

6. Conclusions

This thesis has focused on three main aspects: (i) providing a methodology to develop many-core systems, (ii) adapting the MPI paradigm for such systems, and (iii) implementing new designs to enhance the performance of the system by off-loading specific software tasks to the network interface controller of the system.

As a conclusion, this thesis proposes a message passing interface API on MPSoCs with NoCs interconnections using either shared or distributed memory architectures. Additionally, it has also provided and validated three techniques to improve global computing performance by off-loading software process from the processor of every SoC to the network interface controller.

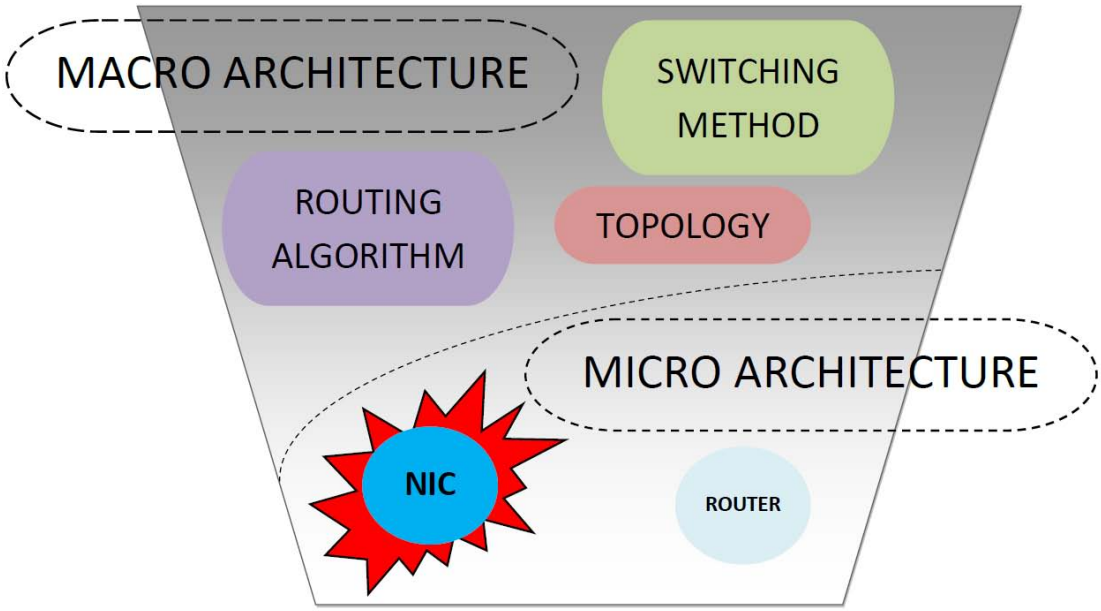


Figure 80 Micro and macro architecture exploration done in the thesis..

This thesis has presented a complete methodology to design MPSoC systems based on NoCMaker and demonstrated implementing a MPSoC on a Statrix II EP2S180

FPGA with 16 Nios -II pr ocessor, FPU and N oC infrastructure runni ng (i) matrix multiplication wi th sub-matrices of 50x 50 a nd 30x 30 e lements amd (ii) a Houg h transform to de tect c ircles with speed-up fa ctors of sp eedup factor of 2. 5, 4 a nd 12 respectively. Results also show the limitation in the implementations of int egrated solutions with t ight li mits in on -chip memory wh en small F PGA de vices are tar geted. The results would be better using devices with large internal memories.

Within this thesis the ocMPI layered software stack to program MPSoC systems has been e volved. That includes MPI la yer, transport layer and low leve l layers such as driver layer.

- Initial developing of performance analysis techniques for MPSoC systems.[102]
- NoCMaker tool Simulation e nvironment for NoC -based MP SoCs, whic h provides early feedback of the performance of the system, has been created and evolved, adding more functionalities and designs.[42]
- Prototyping of (h eterogeneous and homogenou s) NoC-based MP SoCs using different pr ocessors – NIOS-II, whe re a c omplete system with 16 c ores w as shown in chapter 3, Mi crobalze, and L EON together with a proprietary processor called Xentium used in chapter 4, for example.[102]
- Design of s everal network int erfaces for a sm all number of he terogeneous standard busses that includes Altera's Avalon bus, Xilinx Fast Simplest Link bus and PLB bus.
- Create new facilities for network interface controllers that allow the NIC to be master of the bus and, therefore, provide a DMA-like behaviour for the master processor shown in chapter 5, achieving a sp eedup fa ctor of 8 for matrix multiplication tests, while previously we achieved no more than 4.

- Add new facilities to the network interface controller that allow the NIC to implement delivery protocol and, therefore, make the software protocol completely transparent to the upper layers, the master CPU and ultimately to the programmer shown in chapter 5, achieving a speedup factor of 6x when computing the Mandelbrot set in our distributed memory platform, which is a promising result since in a supercomputer such application shows no speedup due to the high rate between communication and computation.[103]
- Design of new facilities to the network interface controller that allow the NIC to enhance MPI synchronization operations such as MPI_INIT shown in chapter 5 that stabilises the time required to initiate a system while having a negligible overhead impact in the size of the system.[101]
- Adapt MPI library for embedded systems over commercial shared memory architecture systems (STHORM and Recore Systems).

This thesis initiates an exploration way over the key topics that must help the NoC-based MPSoCs to achieve a smooth success into the embedded world.

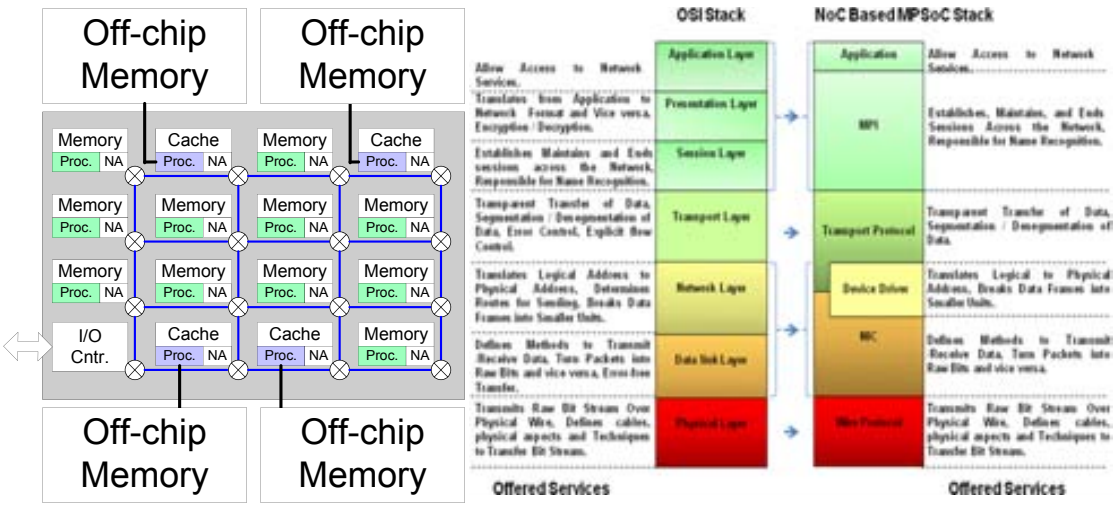


Figure 81 Left) MPSoC architecture **Right**) Software stack for MPSoC.

There are several factors that have an impact on the possible success of NoC-based MPSoCs. However, the focus of this thesis, is centred on what the author considers the main key challenges: (i) design methodology (together with environment and validation designs), (ii) programming models to extract parallelism for the multiprocessor system in a useful and easy way (centred in the vision based upon a MPI programming model as a real alternative to program MPSoCs), and finally, (iii) hardware support oriented to help the system to achieve the previous points.

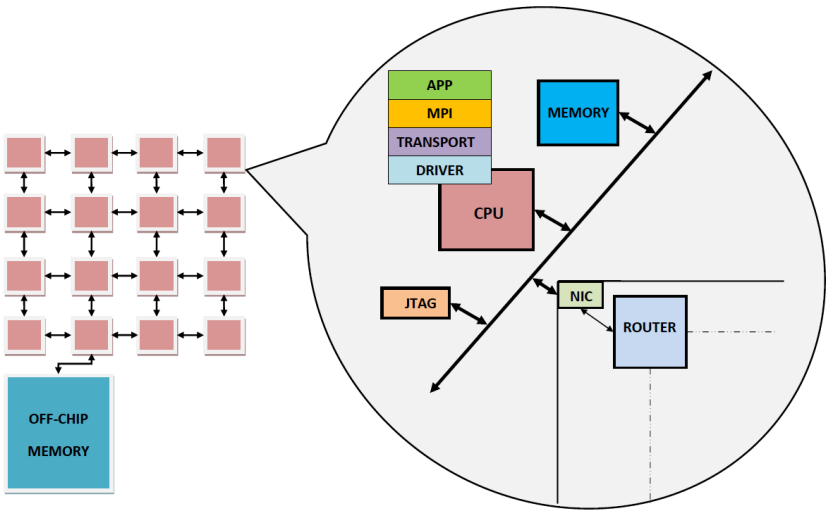


Figure 82 NoC-based MPSoC diagram block.

In summary, the investigation in this thesis has been concentrated in: (i) NoC micro and macro architectures (as shown in Figure 80), (ii) MPSoC memory architecture (Figure 81 left, Figure 82), (iii) software programming stack (Figure 81 right), and (iv) tools for space exploration and simulation framework (Figure 83).

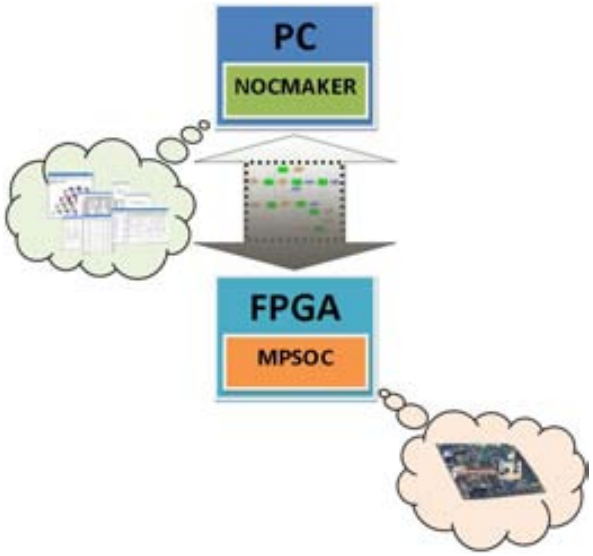


Figure 83 Space exploration on simulation tool connected to a real FPGA to create a MPSoC.

6.1. Open Research

In this thesis, we have tried to show that MPSoCs are key factor for the future of embedded systems. These MPSoC systems will be connected through a (more or less) complex intercommunication system based on a networks-on-chip. These NoCs could have several degrees of complexity, and could present several services including best-effort, traffic guarantees and other traditional quality of services (QoS). The number of open topics for keeping researching in such material is still significant.

However, from the author's point of view, the main challenge to be faced in these days is to find an efficient way to program it.

The hot topic in MPSoCs is the programming models and programming languages. This fact can be proved by looking at the increasing number of new proposals that are appearing recently, like MAPS, or adapted ones (MPI, or OpenMP).

It is obvious that it is really complicated to extract the full potential that a parallel system can offer and a great understanding between the application programmer and the system is needed.

The work presented in this thesis is not an end road. More MPI primitives can be off-loaded to the NIC in the same way that has been done. A first implementation should be the MPI collective primitives, MPI_Broadcast for example. For this case, the Esyncop design offers a stable base to build broadcast services.

However, it is not necessary to be limited to MPI paradigm. Other languages can be studied and some other primitives can be off-loaded.

From personal point of view of the author, it is necessary to improve the programmability of MPSoC systems; however the key factor to do it efficiently must be a combination between HW and SW, in particular, between the NIC and the programming model. The hardware design must be "programming model" oriented and must follow the motto "as easiest to be used by the programming model, the better the system performance could be achieved".

Other areas within the MPSoCs environment that must be objective of research are, from the author point of view, the analysis techniques to study systems performance.

In this thesis, some initial points of such techniques are shown. However, it is necessary to going deeper in the research of such techniques and mechanisms. As the number of processors and components rises, the need for mechanisms, techniques, and tools to analyse the behaviour of such systems increases even faster. Particularly, the Heisenbugs can be a terrible reality in multi and many-core systems almost impossible to solve if the application programmer does not have that kind of techniques. The Heisenbug term appears in computer science to talk about a bug that change its

behaviour depending if someone (typically a developer) is trying to identify, analyse or solve it. In the worst case, the bug can completely disappear when is under analysis.

Other open issues for future research directions can be summarised as:

- NoC services adaptability. Runtime reconfiguration of NoC behaviour is an interesting facility for industry partners. That reconfiguration should include modes for debugging, performance monitoring, power management, fault tolerance,…
- Universal translators. Using some tools, a programmer can create code for a specific language, which he does not know, from another known source language. For instance, OMP2HMPP [98] is an unripe tool that translates OpenMP code into HMPP [99] to be used in GPGPUs. This is a promising road to explore, where several disciplines as artificial intelligence (to detect behaviour patterns within the source code) and compiler design can collaborate together.

REFERENCES

[1] Amdahl. G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, vol. 30 (Atlantic City, N.J.. Apr. 18-20). AFIPS Press, Reston. Va., 1967. pp. 483-485.

[2] A history of innovation [internet]. Available from: http://www.intel.com/content/www/us/en/history/historic-timeline.html

[3] ARM company milestones [intenret]. Available from: http://www.arm.com/about/company-profile/milestones.php

[4] History of IEEE [internet]. Available from: http://www.ieee.org/about/ieee_history.html

[5] Cray time line [internet]. Available from: http://www.cray.com/Assets/PDF/about/CrayTimeline.pdf

[6] Life after smartphones[internet]. Available from: http://sonicsinc.com/blog/2013/06/life-after-smartphones/

[7] Busses, Crossbars and NoCs: The 3 Eras of SoC Interconnect History [internet].Available from: http://info.arteris.com/blog/bid/49081/Busses-Crossbars-and-NoCs-The-3-Eras-of-SoC-Interconnect-History

[8] Tilera's history [internet]. Available from: http://www.tilera.com/about_tilera/tilera_history

[9] Mazumder, P., "Evaluation of On-Chip Static Interconnection Networks," Computers, IEEE Transactions on , vol.C-36, no.3, pp.365,369, March 1987 doi: 10.1109/TC.1987.1676910 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1676910&isnumber=35262

[10] Technology Milestones [internet]. Available from: http://www.altera.com/corporate/about_us/history/abt-history.html

[11] International technology roadmaps for semiconductors [internet]. Available from: http://public.itrs.net/

[12] Sony Xperia z2 specifications [internet]. Available from: http://www.sonymobile.com/global-en/products/phones/xperia-z2/specifications/#tabs

[13] Intel Core i7 processor specifications [internet]. Available from: http://www.intel.com/content/www/us/en/processors/core/core-i7-processor/Corei7Specifications.html

[14] Playstation 4 specifications [internet]. Available from: http://us.playstation.com/ps4/features/techspecs/

[15] Wiltgen, A.; Escobar, K.A.; Reis, A.I.; Ribas, R.P., "Power consumption analysis in static CMOS gates," Integrated Circuits and Systems Design (SBCCI), 2013 26th Symposium on , vol., no., pp.1,6, 2-6 Sept. 2013 doi: 10.1109/SBCCI.2013.6644863 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6644863&isnumber=6644849

[16] A. JANTSCH and H. TENHUNEN. Networks on Chip. Kluwer Academic Publishers, 2003.

[17] L. Benini and G. De Micheli, “Powering networks on chips,” in Proc.ISSS, 2001.

[18] Benini L. et al. Networks on chips: a new SoC paradigm. IEEE Computer, 2002.

[19] W. J. Dally and B. Towles, “Route packets, not wires: on-chip interconnection networks,” in Proc. Design Automation Conf., 2001.

[20] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, J. Tiensyrjä, and A. Hemani, “A network on chip architecture and design methodology,” in Proc. ISVLSI, 2002.

[21] J. L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, May 1988, 532-533

[22] Cray [internet]. Avalable from: http://www.cray.com/Home.aspx

[23]] Flynn M. Some computer organization and their effectiveness. IEEE Trans Comput 1972;C21(9):948–60.

[24] Almasi GS, Gottlieb A. Highly parallel computing. New York: Benjamin Cummings; 1994

[25] Heinrich M, Soundararajan V, Hennessy J, Gupta A. A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols. IEEE Trans Comput 1999;48(2):205–17. doi:10.1109/12.752662.

[26]] Stonebraker M. The case for shared nothing. In: Proceedings of HPTS’1985.

[27] IEEE Xplore digital library [internet]. Available from: http://ieeexplore.ieee.org/Xplore/home.jsp

[28] Dagum, L.; Menon, R.; , "OpenMP: an industry standard API for shared-memory programming," Computational Science & Engineering, IEEE , vol.5, no.1, pp.46-55, Jan-Mar 1998 doi: 10.1109/99.660313

[29] MPI Forum [internet]. Availabble from. http://www.mpi-forum.org

[30] NVIDIA CUDA Compute Unified Device Architecture Programming Guide; NVIDIA: Santa Clara, CA, 2007

[31] A. Munshi, OpenCL Specification Version 1.0, 2008.

[32] Khronos group [internet]. Available from: http://www.khronos.org/registry/cl/.

[33] Multicore Association [internet]. Available from:. http://www.multicore-association.org.

[34] Reinders, J., 2007. Intel Threading Building Blocks.O’Reilly

[35] N. Popovici and T. Willhalm. Putting Intel Threading Building Blocks to work

[36] Phillipe Charles, Christopher Donowa, Kemal Ebcioglu, Christian Grothoff, Alan Kielstra, Christoph Von Praun, Vijay Saraswat, Vivek Sarkar 2005. An object-oriented Approach to Non-Uniform Cluster Computing. ACM OOPSL.

[37] William Thies , Michal Karczmarek , Saman P. Amarasinghe, StreamIt: A Language for Streaming Applications, Proceedings of the 11th International Conference on Compiler Construction, p.179-196, April 08-12, 2002

[38] R.D. Blumofe et. al. Cilk: An eficient multithreaded runtime system. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207{216, Santa Barbara, California, July 1995.

[39] Charles E. Leiserson. The Cilk++ concurrency platform. In 46th Design Automation Conference, San Francisco, CA, July 2009. ACM/EDAC/IEEE.

[40] Parallel Programmability and the Chapel Language Bradford L. Chamberlain, David Callahan, Hans P. Zima. International Journal of High Performance Computing Applications, August 2007, 21(3): 291-312.

[41] Axum programmer's guide. Microsoft corporation.

[42] Castells-Rufas, D.; Joven, J.; Risueño, S.; Fernandez, E. & Carrabina, J. NocMaker: A Cross-Platform Open-Source Design Space Exploration Tool for Networks on Chip INA-OCMC Workshop, Paphos, Cyprus, 2009

[43] PARMA project. URL: PARMA project http://www.parma-itea2.org/

[44] Bellows, P.; Hutchings, B., "JHDL-an HDL for reconfigurable systems," FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on , vol., no., pp.175,184, 15-17 Apr 1998 doi: 10.1109/FPGA.1998.707895 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=707895&isnumber=15334

[45] L. Benini and G. de Micheli, "Networks on chips: A new SoC paradigm," Proceedings of the IEEE Computer, vol. 35, No. 8, pp. 70-78, Jan. 2002.

[46] T. Dorta, J. Jimenez. JL. Martin, U. B. A. A. Overview of FPGA-Based Multiprocessor Systems Reconfigurable Computing and FPGAs, International Conference on, IEEE Computer Society, 2009, 0, 273-278.

[47] G. Mplemenos, I. P. MPLEM: An 80-processor FPGA Based Multiprocessor System Field-Programmable Custom Computing Machines, Annual IEEE Symposium on, IEEE Computer Society, 2008, 0, 273-274.

[48] Tseng, C. & Chen, Y. Design and Implementation of Multiprocessor System on a Chip (MPSoC) Based on FPGA 2009.

[49] Wang, Z. & Hammami, O. External DDR2-Constrained NOC-Based 24-Processors MPSOC Design and Implementation on Single FPGA Design and Test Workshop, 2008, 193-197

[50] Tian, G and Hammami, O., "Performance measurements of synchronization mechanisms on 16PE NoC Based Multi-Core with dedicated synchronization and Data NoC" in Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on.

[51] Chalamalasetti, S.R., Vanderbauwhede, W., Purohit, S. and Margala, M. (2009) *A low cost reconfigurable soft processor for multimedia applications: design synthesis and programming model.* In: 2009 International Conference on Field Programmable Logic and Applications. IEEE Computer Society, Piscataway, N.J., USA, pp. 534-538

[52] Chalamalasetti, S.R., Purohit, S., Margala, M. and Vanderbauwhede, W. (2009) *MORA - an architecture and programming model for a resource efficient coarse grained reconfigurable processor.* In: 2009 NASA/ESA Conference on Adaptive Hardware and Systems, 29 July 2009 - 1 Aug. 2009, San Francisco, CA, USA. IEEE Computer Society, Piscataway, N.J., USA, pp. 389-396

[53] Gohringer, D.; Hubner, M.; Hugot-Derville, L.; Becker, J.; , "Message Passing Interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC," Embedded Computer Systems (SAMOS), 2010 International Conference on , vol., no., pp.357-364, 19-22 July 2010

[54] Yujia Jin et al. ; "An automated exploration framework for FPGA-based soft multiprocessor systems," Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on , vol., no., pp.273-278, Sept. 2005

[55] Hristo Nikolov; Todor Stefanov; Ed Deprettere; , "Efficient Automated Synthesis, Programing, and Implementation of Multi-Processor Platforms on FPGA Chips," Field Programmable Logic and Applications, 2006. FPL '06. International Conference on , vol., no., pp.1-6, Aug. 2006.

[56] Atienza, D.; Del Valle, P.G.; Paci, G.; Poletti, F.; Benini, L.; De Micheli, G.; Mendias, J.M.; , "A fast HW/SW FPGA-based thermal emulation framework for multi-processor system-on-chip," Design Automation Conference, 2006 43rd ACM/IEEE , vol., no., pp.618-623

[57] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC", presented at VLSI Signal Processing, 2005, pp.169-182.

[58] Carara, E.; Oliveira, R.; Calazans, N.; Moraes, F. "HeMPS - A Framework for NoC-Based MPSoC Generation". In: ISCAS'09, 2009.

[59] J. Joven et al. , "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 2008). Toulouse, France, February 13-15, 2008.

[60] Altera, Inc. Applying the Benefits of Network on a Chip Architecture to FPGA System Design. Version 1.0, January 2011.

[61] Duato, J., Yalmanchili, S. Interconnection Networks, An Engineering Approach. Morgan Kaufman Publishers, Elsevier Science, 200

[62] D. Castells-Rufas, J. Joven, S. Risueño, E. Fernandez, J. Carrabina, T. William, H. Mix. "MPSoC Performance Analysis with Virtual Prototyping Platforms". PSTI, San Diego, USA, September, 2010.

[63] Atienza D, Angiolini F, Murali S, Pullini A, Benini L, De Micheli G. Network-on-chip design and synthesis outlook. Integration 2008:340–59

[64] Altera, Inc., Nios II Processor Reference handbook document, version: version 6.0.0, 2006

[65] Altera, Inc., AVALON Bus specification–reference manual, version 2.0, January 2002.

[66] Altera, Inc., Using Nios II floating-point custom instructions tutorial, February 2010.

[67] Available from: <http://www.alterawiki.com/wiki/Configurable_FPU>.

[68] Bosschere KD, Luk W, Martorell X, Navarro N, O'Boyle MFP, Pnevmatikatos DN, et al. High-performance embedded architecture and compilation roadmap, Transactions on HiPEAC I (s2007). p. 5–29.

[69] Duda RO, Hart PE. Use of the Hough transformation to detect lines and curves in picture. In: Proceedings of commun. ACM, 1972. p. 11–5.

[70] Psota J, Agarwal A. RMPI: message passing on multicore processors with on-chip interconnect. Lect Notes Comput Sci 2008;4917:22.

[71] Saint-Jean N, Benoit P, Sassatelli G, Torres L, Robert M. MPI-based adaptive task migration support on the HS-scale system. In: 2008 IEEE computer society annual symposium on VLSI (ISVLSI), 2008. p.105–10.

[72] Multicore Association Communication API Specification V1.063. Available from: <http://www.multicore-association.org>.

[73] Brunst H, Hoppe HC, Nagel WE, Winkler M. Performance optimization for large scale computing: the scalable VAMPIR approach. In: International conference on computational science, 2001. p. 751–60.

[74] Nagel WE, Arnold A, Weber M, Hoppe HC, Solchenbach K. VAMPIR: visualization and analysis of MPI resources. Supercomputer 1996;12:69–80

[75] Joven J. A lightweight MPI-based programming model and its HW support for NoC-based MPSoCs. Ph.D. Forum DATE, IEEE/ACM Design, Automation and Test in Europe (DATE'09), Nice, France, April 2009.

[76] Eduard Fernandez-Alonso, David Castells-Rufas, Jaume Joven, Jordi Carrabina. "Embedding MPI in Many-Soft-Core Processors"; In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2013)

[77] Gohringer, D.; Hubner, M.; Hugot-Derville, L.; Becker, J.; , "Message Passing Interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC," Embedded Computer Systems (SAMOS), 2010 International Conference on , vol., no., pp.357-364, 19-22 July 2010

[78] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M.Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall: "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation"; In Proc. of 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97-104, Sept. 2004.

[79] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, A. Lumsdaine: "Open MPI: A High Performance, Heterogenous MPI"; In Proc. of Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Barcelona, Spain, September 2006.

[80] W. Gropp, E. Lusk, A. Skjellum: "Using MPI: Portable Parallel Programming with the Message-Passing Interface"; MIT Press, 1999.

[81] M. Saldana, P. Chow: "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs"; In Proc. of the 16th International Conference on Field-Programmable Logic and Applications (FPL 2006), Madrid, Spain, 2006.

[82] P. Mahr, C. Lörchner, H. Ishebabi, C. Bobda: "SoC-MPI: A flexible Message Passing Library for Multiprocessor Systems-on-Chips"; In Proc. of IEEE International Conference on Mexico, ReConFigurable Computing and FPGAs (ReConFig'08), Cancun, December 2008

[83] Catrene COBRA project. Available on: http://www.catrene.org/web/downloads/profiles_catrene/CATRENE%20project%20profile-CA104-outCO%20(18-7-11).pdf

[84] Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology. CEA, STMicroelectronics, Nov. 2010

[85] Recore systems, Xentium processor. URL: Xentiumhttp://www.recoresystems.com/products/xentium-vliw-dsp-ip/

[86] Rashti, M. & Afsahi, A. Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects 22nd International Symposium on High Performance Computing Systems and Applications (HPCS 2008), 2008, 95-101.

[87] D. Castells-Rufas, J. Joven, J. Carrabina, "A Validation and Performance Evaluation Tool for ProtoNoC". International Symposium on System-on-Chip 2006 (SOC2006). Tampere, Finland, November 13-16, 2006

[88] Book Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©2003 ISBN:1558607242

[89] Wen Su; Ling Wang; Menghao Su; Su Liu, "A Processor-DMA-Based Memory Copy Hardware Accelerator," Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on , vol., no., pp.225,229, 28-30 July 2011 doi: 10.1109/NAS.2011.15. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6005465&isnumber=6005426

[90] Jimack, P K & Touheed, N, An Introduction to MPI for Computational Mechanics. In Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, ed. B.H.V. Topping (Saxe-Coburg Publications), pp.24-45, 1999.

[91] J. Duato, S. Yalamanchili, N. Lionel. Interconnection Networks: An Engineering Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2002.

[92] How to sound like a Parallel Programming Expert Part 1: Introducing concurrency and parallelism [internet]. Available from software.intel.com/en-us/articles.

[93] Altium, TSK300A risc processor. Documentation. Available from: http://techdocs.altium.com/display/ADRR/TSK3000A+Pipeline

[94] Bellows, P.; Hutchings, B., "JHDL-an HDL for reconfigurable systems," *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on* , vol., no., pp.175,184, 15-17 Apr 1998 doi: 10.1109/FPGA.1998.707895 , URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=707895&isnumber=15334

[95] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1137–1142. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228568

[96] Torquati Massimo, Bertels Koen, Karlsson Sven, Pacull François; "The STHORM platform" , Smart Multicore Embedded Systems book. 2014; Springer New York, New York, NY.

[97] Computer Architecture: A Quantitative Approach, 4th Edition (27 September 2006) by John L. Hennessy, David A. Patterson

[98] Albert Saà-Garriga, David Castells-Rufas, Jordi Carrabina . "OMP2HMPP: HMPP Source Code Generation from Programs with Pragma Extensions " Conference Proceeding, 01/2014; In proceeding of: High Performance Energy Efficient Embedded Systems (HIP3ES 2014).

[99] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. In Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), 2007.

[100] Ecomunicat electronics.Available from: http://www.ecomunicat.com/index.html

[101] Eduard Fernandez-Alonso, D. Castells-Rufas, J. Joven,J. Carrabina. Embedding MPI in Many-Soft-Core Processors**.** High Performance Energy Efficient Embedded Systems, HIP3ES, 2013

[102] Eduard Fernandez-Alonso, D Castells-Rufas, J Joven, J Carrabina. Development process for clusters on a reconfigurable chip. Computers & Electrical Engineering 38 (3), 756-771 2012.

[103] Eduard Fernandez-Alonso, D. Castells-Rufas, J. Carrabina. Enhancing MPI delivery protocol in NoC-based MPSoC system. International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies, HEART, 2012

AUTHOR'S RELEVANT PUBLICATIONS

Articles directly referred in the PhD

1. **Eduard Fernandez-Alonso**, D. Castells-Rufas, J. Joven,J. Carrabina. **Embedding MPI in Many-Soft-Core Processors.** High Performance Energy Efficient Embedded Systems, HIP3ES, 2013
2. **Eduard Fernandez-Alonso**, D. Castells-Rufas, J. Carrabina. **Enhancing MPI delivery protocol in NoC-based MPSoC system**. International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies, HEART, 2012
3. **Eduard Fernandez-Alonso**, D Castells-Rufas, J Joven, J Carrabina. **Development process for clusters on a reconfigurable chip**. Computers & Electrical Engineering 38 (3), 756-771 2012.
4. **E Fernandez-Alonso**, D Castells-Rufas, J Joven, J Carrabina. **Survey of NoC and Programming Models Proposals for MPSoC**. International Journal of Computer Science 2012.
5. **E Fernandez**, D Castells-Rufas, S Risueño, J Joven, J Carrabina. **Hybridising NiC/NoC switching techniques**. Conference on Design of Integrated Circuits, DCIS 2010.
6. D Castells-Rufas, J Joven, S Risueño, **E Fernandez**, J Carrabina **NocMaker: A cross-platform open-source design space exploration tool for networks on chip**. INA-OCMC Workshop, Paphos, Cyprus 2009
7. J Joven, D Castells-Rufas, S Risueño, **E Fernandez**, J 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 2009

Articles related to the PhD

1. D Castells-Rufas, **Eduard Fernandez-Alonso**, J Carrabina. **Performance analysis techniques for multi-soft-core and many-soft-core systems**. International Journal of Reconfigurable Computing 2012, 2
2. David Castells-Rufas, **Eduard Fernandez-Alonso**, and Jordi Carrabina. **Trace generation in many-soft-cores**. International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies, HEART, 2012
3. **Eduard Fernandez-Alonso**, D Castells-Rufas, S Risueño, J Carrabina, J Joven. **A NoC-based multi-{soft} core with 16 cores**. Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International conference, 2010.

Other articles not directly related with the thesis topics

1. J Joven, A Bagdia, F Angiolini, P Strid, D Castells-Rufas, **E Fernandez**. **QoS-driven Reconfigurable Parallel Computing for NoC-based Clustered MPSoCs**. ... Industrial Informatics, IEEE 2012

2. D Castells-Rufas, **E Fernandez-Alonso**, J Carrabina, J Joven. **Sharing FPUs in many-soft-cores**. Field-Programmable Technology (FPT), 2011 International Conference on, 1-6 2011
3. D Castells-Rufas, J Joven, S Risueño, **E Fernandez**, J Carrabina, T William, H Mix. **MPSoC performance analysis with virtual prototyping platforms**. Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, 2010
4. D Castells-Rufas, S Risueño, **E Fernandez**, J Carrabina, J Joven. **Instruction Set Extensions to Reduce Latency in Soft-Core Clusters**. Conference on Design of Integrated Circuits, DCIS 2010.

CURRICULUM VITAE

PERSONAL INFORMATION

Name	**FERNANDEZ ALONSO, EDUARD**
E-mail	**Eduard.Fernandez.Alonso@gmail.com** **Eduardo.Fernandez@uab.es**
Date of birth	2 May 1980

WORK EXPERIENCE

• Dates (from – to)	From November 2012
• Name and address of employer	Centre d'Accesibilitat i Intel·ligència Ambiental de Catalunya (CaiaC)
• Type of business or sector	Microelectronics and embedded systems
• Occupation or position held	Research support technician
• Main activities and responsibilities	The main activities and responsibilities include developing and programming MPSoCs. Currently, I am also involved on European projects (FP7).
• Dates (from – to)	From June 2012 to November 2012
• Name and address of employer	Recore Systems, Enschede, The Netherlands
• Type of business or sector	Microelectronics and embedded systems
• Occupation or position held	Internship
• Main activities and responsibilities	The main activity was to investigate programming of multi-core SoC architectures using Recore's multi-core systems, and, therefore, the focus of the main activities done at Recore was to implement a subset of the standard MPI library for a multi-core system based on Xentium and NoC technology. I had the responsibility to build a new NoC-based MPSoC system using Xentium processors and additionally, to develop a new software stack to program such processor.
	.
• Dates (from – to)	From July 2010
• Name and address of employer	Centre d'Accesibilitat i Intel·ligència Ambiental de Catalunya (CaiaC)
• Type of business or sector	Microelectronics and embedded systems
• Occupation or position held	Research support technician
• Main activities and responsibilities	The main activities and responsibilities include developing and programming MPSoCs. Currently, I am also involved on European projects, for instance COBRA project where I worked adapting a lightweight MPI library over ST-Microelectronics P2012/STHORM Platform.
• Dates (from – to)	From May 2008 to July 2010
• Name and address of employer	Cephis, Universitat Autònoma de Barcelona (UAB)
• Type of business or sector	Microelectronics and embedded systems
• Occupation or position held	Research support technician
• Main activities and responsibilities	The main activities and responsibilities include developing networks on chip for MPSoCs, programming in Java (since a part of the work done was developing a design exploration tool for NoCs named NoCMaker), in C, in hardware description languages (Verilog, VHDL, JHDL), and designing and synthesizing systems on a FPGA board.

EDUCATION AND TRAINING

• Dates (from – to)	From 2009 to 2010
• Name and type of organisation providing education and training	Universitat Autònoma de Barcelona (UAB)
• Title of qualification awarded	Master on Global Business Management

• Dates (from – to)	From 2008 to 2009
• Name and type of organisation providing education and training	Universitat Autònoma de Barcelona (UAB)
• Title of qualification awarded	Master on Micro and Nano Electronics
• Dates (from – to)	From 2001 to 2008
• Name and type of organisation providing education and training	Universitat Autònoma de Barcelona (UAB)
• Title of qualification awarded	Computer Science
Additional information	Member of Program Committee of JCE (Jornadas de Computación Empotrada) which is part of SARTECO (Sociedad de Arquitectura y Tecnológia de Computadores). Professor at Master of “Tecnologies de la informació geográfica”, Universitat Autònoma de Barcelona.