UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

TU/e technische universiteit eindhoven

Fast modelling and analysis of NoC-based MPSoCs

M. Teresa Medina León

Master of Science thesis

Project period: November 2005 – September 2006

Eindhoven University of Technology (TU/e)
Department of Electrical Engineering
Capacity group: Information and Communication Systems
Chair: Electronic Systems

Supervisors at TU/e:
Prof. Dr. H. Corporaal
Dr. Ir. B. Theelen

Supervisors at ULPGC:
Prof. Dr. F. Tobajas
Ir. V. Reyes

The Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of M.Sc. thesis or practical training reports

To my parents Ricardo and Teresa,

and my sister Virginia

Abstract

Network-on-chip (NoC) based multi-processor systems-on-chip (MPSoC) are emerging as suitable processor platforms for multi-media applications. The MiniNoC project investigates the design and implementation of multimedia systems based on NoCs. Therefore, different design options and trade offs have to be explored between different communication protocols and configurations. Unfortunately the simulations of the MiniNoC platform are extremely slow and the effort required to create its components is very high due to the low abstraction level (synthesizable SystemC RTL) chosen for its modeling. This makes the current MiniNoC platform model unsuitable for analysis and exploration activities, where many simulations have to be carried out following an iterative approach.

As a solution for these problems Transaction Level Modeling (TLM) is being proposed as the next abstraction level above RTL. Since TLM models are faster to develop and simulate faster than RTL models, designers can explore and verify the system and make design decisions early in the design flow, which improves quality and reduces design time. In order to further improve the benefits of TLM and to ease the modeling of multiprocessor systems, the CASSE environment has been developed. CASSE is a system-level modeling and simulation environment that provides a seamless flow from application specification to system implementation, via architectural modeling, function to architecture mapping and performance analysis.

This Master Thesis investigates how the CASSE tool suits the modeling and simulation of NoC-based MPSoCs. For that purpose, a Mixed-Level model (RTL and TLM) of the MiniNoC platform is implemented using the tool. Moreover a multimedia application is mapped on the resulting platform model in order to compare the modeling effort, simulation speed and accuracy between the original MiniNoC platform and the CASSE Mixed-Level model.

Contents

Chapter 1 1
Introduction 1
1.1 Technical context 1
1.2 Objectives 3
1.3 Outline of the Document 3

Chapter 2 5
MiniNoC Platform 5
2.1 Introduction 5
2.2 Architecture 5
2.2.1 *mMIPS* 6
2.2.2 MEMDEV and Network Interface 6
2.2.3 *MiniNoC* network 8
2.3 C libraries for the MiniNoC 9
2.3.1 C message passing library (*stdcomm*) 9
2.3.2 *mMIPS* mtools library (mtools) 9
2.3.3 LCC C Compiler 10
2.4 Running an application 10

Chapter 3 13
CASSE tool 13
3.1 Introduction 13
3.2 Design methodology 13
3.2.1 Application modeling 14
3.2.2 Architectural modeling 14
3.2.3 Mapping and execution 15
3.2.4 Analysis 15
3.2.5 Refinement 15

Chapter 4 17
CTAP tool 17
4.1 Introduction 17
4.2 Functionality 17

Chapter 5 19
Applications 19
5.1 Introduction 19
5.2 JPEG application 19

Chapter 6 21
MiniNoC Modifications 21
6.1 Introduction 21
6.2 TIME_STAMP instruction 21
6.3 New *stdcomm* library 22
6.4 JPEG decoder 22
6.5 Memory extension 22

Chapter 7 23
CASSE Mixed-Level model 23
7.1 Introduction 23
7.2 Application modeling 23
7.2.1 *Comm* class 24
7.2.2 Gossip application 25
7.2.3 JPEG modifications 26
7.3 Architecture modeling 26
7.3.1 Creating the 4-tiles platform with CASSE 27
7.3.2 Adapting and integrating the *mNoC* component 28
7.3.3 CASSE Mixed-Level model 30
7.4 Mapping 32
7.5 Time annotations and analysis 33
7.5.1 Alternatives for timing annotation 33
7.5.2 Timing annotations in CASSE 34
7.5.3 *mMIPS* Instrumentation 35
7.5.4 Applying the CTAP tool 35
7.5.5 Correcting the compiler error 37
7.5.6 Results 38
7.6 Conclusions 38

Chapter 8 41
Comparison 41
8.1 Introduction 41
8.2 Metrics definition 41
8.3 Metrics evaluation 42
8.3.1 Performance of JPEG decoder 42
8.3.2 Latency of processing each macro block 48
8.3.3 Total latency of sending one specific message 56
8.4 Conclusions 60

Chapter 9 61
Conclusion and future work 61
9.1 Conclusions 61
9.2 Recommendations 62
9.3 Future work 62

Bibliography 65
Abreviation 67
Appendix A 69

List of Figures

Chapter 1
Figure 1.1: SoC design flow 2

Chapter 2
Figure 2.1: *MiniNoC* Platform 6
Figure 2.2: MEMDEV module accesses RAM or NI depending on the address 7
Figure 2.3: Meaning of the bits in the NI control word (device address: 0x80000004) .. 7
Figure 2.4: Four routers (xYyY), four miniMipses (dp_xXyY) and the data lines that connect them 8
Figure 2.5: Process followed to run an application in the *MiniNoC* platform 10
Figure 2.6: Extract of the *main_net.cpp* file 11
Figure 2.7: View of the VCD file with the waveform viewer 11

Chapter 3
Figure 3.1: CASSE design flow 14

Chapter 4
Figure 4.1: CTAP tool process 18
Figure 4.2: Example of timing annotation applying CTAP 18

Chapter 5
Figure 5.1: JPEG decoding process 19

Chapter 7
Figure 7.1: Class hierarchy 24
Figure 7.2: Extract of the *sc_send()* primitive code of the *comm* class 25
Figure 7.3: Gossip for 2x2 network 25
Figure 7.4: Application description file 26
Figure 7.5: Mixed-Level model 27
Figure 7.6: Modeling the *MiniNoC*'s PTile with CASSE 27
Figure 7.7: Adaptors connect abstract transaction-level models to signal-level modules 28
Figure 7.8: Adaptor Implementation 29
Figure 7.9: Extract of the *Ninet.h* file 30
Figure 7.10: NOC connections 30
Figure 7.11: CASSE Mixed-Level model 31
Figure 7.12: Architecture description file 32
Figure 7.13: Mapping description file 32
Figure 7.14: Input image *surfer.jpg* 34
Figure 7.15: Annotation example of the *mMIPS* Instrumentation model 35
Figure 7.16: Process followed to apply CTAP to the JPEG decoder 36
Figure 7.17: Annotation example of the model 36
Figure 7.18: Annotation example of the modified model 37

Chapter 8
Figure 8.1: Output for the JPEG application for the surfer.jpg 43
Figure 8.2: CASSE output for the annotated model after decoding the *surfer.jpg* 44

Figure 8.3: Accuracy evaluation 47
Figure 8.4: Simulation time comparison 47
Figure 8.5: Simulation speed comparison 48
Figure 8.6: Extract of the standard output 49
Figure 8.7: Comparison of the macro block time processing for the *surfer* image 51
Figure 8.8: Comparison of the latency of the macro blocks for the *surfer* image 52
Figure 8.9: Comparison of the macro block time processing for the *pyramid* image 53
Figure 8.10: Comparison of the latency of the macro blocks for the *pyramid* image 54
Figure 8.11: Comparison of the macro block time processing for the *surfer2* image 55
Figure 8.12: Comparison of the latency of the macro blocks for the *surfer2* image 56
Figure 8.13: Nodes used in the application 57
Figure 8.14: *Test* application performed for 1 packet message 57
Figure 8.15: P Tiles used in the application 58
Figure 8.16: Comparison of the latencies of the *sc_ send()* primitive in both models 59
Figure 8.17: Comparison of the latencies of the *sc_ receive()* primitive in both models 60

Chapter 9
Figure 9.1: Modules proposed for future work 63

Appendix A
Figure A.1: Timing progress 69

List of Tables

Chapter 7

Table 7. 1: Accuracy of the different annotated models.. 38

Chapter 8

Table 8.1: Comparison for different size images 46
Table 8.2: Latency of processing each macro block for the RTL for the *surfer* image . 50
Table 8.3: Latency of processing each macro block for the CASSE Mixed-Level model for the *surfer* 51
Table 8.4: Comparison of the macro block's latency for the *surfer* image 52
Table 8.5: Latency of processing each macro block for the RTL model for the *pyamid* image 53
Table 8.6: Latency of processing each macro block for the Mixed-Level model for the *pyamid* image.. 53
Table 8.7: Comparison of the macro block's latency for the *pyramid* image 54
Table 8.8: Latency of processing each macro block for the RTL for the *surfer2* image 55
Table 8.9: Latency of processing each macro block for the CASSE Mixed-Level model for the *surfer2* 55
Table 8.10: Comparison of the macro block's latency for the *surfer2* image 56
Table 8.11: Latency of the *sc_send()* primitive for different size message at the both platforms.. 58
Table 8.12: Relative error introduced in the *sc_send()* primitive latency of the Mixed-Level model 59
Table 8.13: Latency of *sc_receive()* for different size message at the both platforms.. 59
Table 8.14: Relative error introduced in the *sc_receive()* primitive latency of the Mixed-Level model 60

Chapter 1

Introduction

1.1 Technical context

The Systems-on-a-Chip (SoCs) are a new category of systems which have emerged during recent years. In such systems processor cores and other system components available as intellectual properties (IPs) are integrated on a single chip. These IPs (CPUs, DSPs, memories, peripherals, etc.) incorporate more reusability [1-2] and flexibility in the design since they can be quickly customized and integrated into multiple design projects.

Currently, SoCs are increasing in complexity [3-4] and in recent times the designers have started employing an increasing number of Multi-Processor System-on-chip (MP-SoC) platforms combining several programmable devices. Owing to the high number of cores that require communications between them it is not feasible to use a single shared bus or a hierarchy of buses. To address this problem Network-On-Chip (NoC) has been proposed [5]. It consists of a set of routers that compose a network, which allows all the nodes connected to it to be able to communicate each others.

Designing a System-on-a-Chip includes the development of software and hardware. The constraints of the market force the development cycle of a chip to remain as short as possible, and it is not acceptable to wait for the chip to physically exist to start the development of the corresponding software. The development of the software has therefore to be carried out, or at least started, on a simulator.

The synthesizable approach (Register Transfer Level) does not make it possible to start the development of the software before the code for the synthesizable version of the hardware is written. Moreover, the slowness of the simulation does not allow scaling up. The simulation of the synthesizable model is too slow because it uses a level of abstraction (RTL) which includes too many details of implementation, of protocols, which are not relevant for the software.

To overcome these limitations, systems designers have raised the abstraction level of system models and allow system-level modeling [6]. A new level of abstraction called Transaction Level Modeling (or TLM) has been introduced in the design flow in which only what is necessary for the software to run is modelled [7]. The high speed of simulation of these Transactions Level Models allows early development and verification of hardware dependent application software [8]. Timing details can be incorporated into these models to allow performance estimation and architecture explorations. As a result, the models give an early estimate of the system characteristics before committing to RTL development.

At this level, a platform is a set of modules connected by communication channels. The communication channels transport transactions, which are an atomic exchange of data between two modules. A data transaction can be a single word, a series of words or a complex data structure.

TLM is a concept independent of language. However, to implement and refine TLM models, it is helpful to have a language like SystemC [9-13] whose features support independent refinement of functionality and communication that is crucial to efficient TLM development.

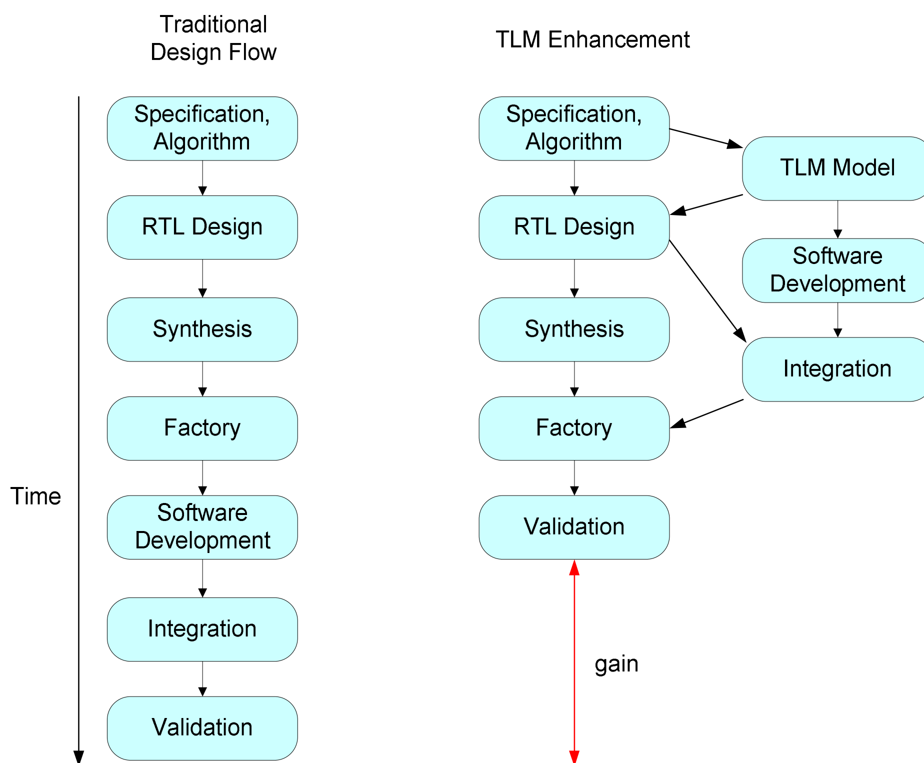


Figure 1.1: SoC design flow

SystemC is a C++ library aimed specifically at system level modeling. It has all the benefits that C++ possesses – it is an object oriented design language that makes full use of data encapsulation and generic programming concepts. SystemC 2.1 facilities the development of Transaction Level Models since provides modules, signals to model the low-level communications and synchronizations of the system, and a notion of simulation time [14-15].

1.2 Objectives

The *MiniNoC* project consists of a very simple *MP-SoC* platform (implemented in FPGA) with four processor nodes interconnected with a *NoC* network component [16]. The components of the *MiniNoC* platform are described in a RTL specification, thus the platform is synthesizable and 100% accurate. However, there are also disadvantages: it needs a high modeling effort and simulations are extremely slow. This makes the current model unsuitable for performance analysis and exploration activities - where many simulations have to be carried out following an iterative approach. In order to improve the simulation speed and to ease the modeling of multiprocessor systems, the CASSE environment can be used. CASSE is a system-level modeling and simulation environment that provides a seamless flow from application specification to system implementation, via architectural modeling, function to architecture mapping and performance analysis [18].

The main objective of this Master Thesis is to adapt and integrate the *MiniNoC* components in CASSE in order to enable an evaluation of performing *NoC*-based *MPSoC* modeling with CASSE. To this end, a CASSE Mixed-level model will be develop where the *MiniNoC* components will be slightly adapted using CASSE compliant transaction-level communication protocols, which will considerably improve the simulation speed although keeping a good accuracy level. A multimedia application will be mapped to the resulting model in order to make a comparison between the CASSE Mixed-level model and the *MiniNoC* RTL-level SystemC specification. For that purpose, the application will be adapted to use the programming model provided by CASSE. The results of comparison will be analyzed to make an evaluation of the simulation speed, accuracy, and modeling effort, and also analyze whether *CASSE* improves the design space exploration.

1.3 Outline of the Document

This document is divided into three parts:

It starts with Chapter 2, "*MiniNoC platform"* where the architecture and functioning of the *MiniNoC* platform is described, the starting point of this project. Chapter 3, "*CASSE tool*" introduces the features of CASSE, a system-level modeling and simulation environment applied to perform our new platform. It follows with Chapter 4, "*CTAP tool*" that describes briefly the functionality of the CTAP tool, used to annotate timing information into untimed models. Finally, Chapter 5, "*Applications*" introduces the JPEG decoder, which is the application that has been chosen for performing the comparison.

The second part starts with Chapter 6, "*MiniNoC* Modifications" that describes the modifications performed to the *MiniNoC* platform. The following Chapter 7, "*CASSE Mixed-Level model"* presents the steps followed to develop the new platform, the CASSE Mixed-Level model. Finally, Chapter 8, "*Comparison*" shows the results obtained from the metrics done to each platform.

The last part contains Chapter 9, "*Conclusion and future work*", which presents the conclusions based on the obtained results, and some future work proposed to continue with this Master Thesis.

Chapter 2

MiniNoC Platform

2.1 Introduction

This chapter will describe the architecture and functioning of the *MiniNoC* platform. It starts with a brief overview of the architecture of the platform, after that it explains in detail the functionality of each module and how the communication takes place. The chapter ends with a section which explains how an application is run in the *MiniNoC*.

2.2 Architecture

The *MiniNoC* platform is composed of four *mMIPS* processors [16] displayed in four nodes that communicate with each other via the *mNoC* network which contains the routers. This Network-On-Chip consists of a torus network with two nodes wide and two nodes high that follows an E-ecube routing (*Figure 2.1*). The implementation of the NoC is in C++ using SystemC libraries [14], and all the example applications to test the functionality of the system have been written in C.

All the *mMIPS* processors access the network by communications with the bidirectional module called the *Network Interface* (NI). An intermediate module is between the processor and the network interface: the MEMDEV. This module controls the data memory of the *mMIPS*, reads and writes to the RAM memory and performs the communications with the network interface. Two nodes communicate with each other through the network by passing messages which are implemented by the *sc_send()* and *sc_receive()* software functions. These functions are implemented in the *stdcomm* library.

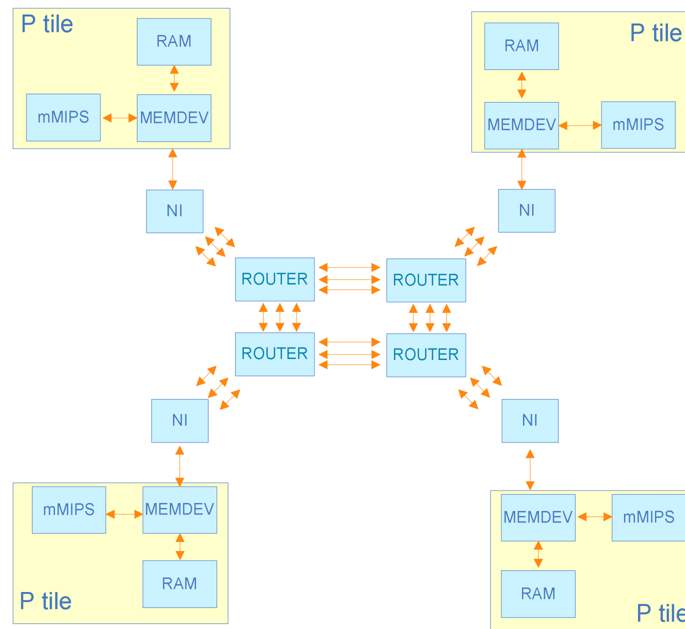


Figure 2.1: ***MiniNoC*** **Platform**

2.2.1 *mMIPS*

The MIPS processor, designed in 1984 by researchers at Stanford University, is a RISC (Reduced Instruction Set Computer) processor. Compared with their CISC (Complex Instruction Set Computer) counterparts (such as the Intel Pentium processors), RISC processors typically support fewer and much simpler instructions.

The *mMIPS* (mini MIPS) processor is a simplified version of the MIPS processor [17]. It is a pipelined system performed in synthesizable SystemC code. Compared to the MIPS it has a reduced instruction set, which means that some operations need to be done in software and therefore more execution time is needed.

Due to the limited instruction set the following operations are done in software:

- All floating point operations
- Multiply, divide, modulo
- Variable distance shifts
- Partial-word operations

2.2.2 MEMDEV and Network Interface

All the *mMIPS* processors communicate with other nodes through the network using the Network Interface (NI). Since this module is memory mapped it can be accessed through specific memory locations. The NI is controlled in *mMIPS* assembler by appropriate stores/loads to/from these memory locations. Another module, MEMDEV, controls the data memory of the *mMIPS* and, based on the requested memory address either read/writes the RAM memory (for regular addresses) or performs appropriate communications with network interface (for device addresses).
This configuration is described in the following figure:

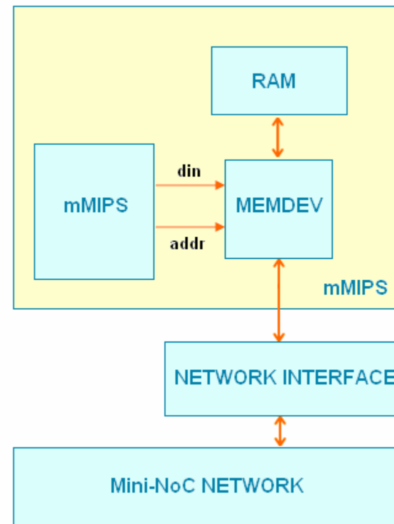


Figure 2.2: MEMDEV module accesses RAM or NI depending on the address

The MEMDEV module recognizes two addresses assigned to the NI: 0x80000000 and 0x80000004. The first address (data word address) is associated with NI data while the second word (control word address) is used for NI control. The reading and writing from the data word results in the reading/writing from internal buffers of the NI. The read/write operations into the data word are always non-blocking, which means that regardless the state of the NI they read/write the NI buffers. However, depending on the state of NI the read data may be invalid, or the written data can overwrite the word in the send buffer, which was not sent yet. To avoid these kinds of problems there are control signals in the NI which are asserted when the data is ready to read or when the buffer is free to write. To monitor the status of the NI, the control word of the NI can be accessed by the memory address 0x80000004. The meaning of the bits in the word is explained in *Figure 2.3*:

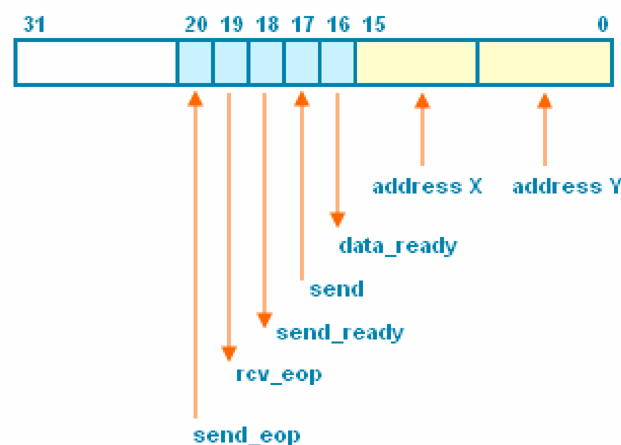


Figure 2.3: Meaning of the bits in the NI control word (device address: 0x80000004)

According to *Figure 2.3* and to the arrows on it note that some of the bits in the control word can only be written (0-15, 17 and 20 - they control the behaviour of NI) while some can only be read (16, 18 and 19 - they report the status of NI). Also note that bit from 21 to 31 are free and could be used for other purposes.

The status bits include:

- *data ready* (bit 16) - new data has arrived and is ready for reading. This bit will be automatically cleared after the data word has been read.
- *send ready* (bit 18) - previous data has been sent. NI is ready to send, data word can be safely written.
- *received end of packet* (bit 19) - multiword packets can be read from NI by reading consecutive 32-bit words. If this bit is active together with data ready, it means that the new data to be read is the last word of the packet.

The control bits:

- *address bits* (bits 0-15) - used to write the destination address of the packet (X distance - bits [15:8], Y distance - bits [7:0]).
- *send bit* (bit 17) -writing 1 to send bit triggers sending the data previously written to the data word to the address present at the address bits of the control word.
- *send end of packet* (bit 20) - asserting bit 20 together with the send bit (17) means that the word written to data word is the last word of the packet and instructs NI to close the packet.

2.2.3 *MiniNoC* network

The network that connects the *mMIPS* processors on the NoC is a *torus network* with *E-cube* routing two nodes wide and two nodes high.

In the *torus* topology, a network node is connected to its immediate neighbours in both dimensions. At edges of the network, the connections wrap around and connect the last router in the given dimension with the first. See *Figure 2.4*.

In the E-cube routing each packet in the network is first routed along the X dimension, until it reaches a router with the X address equal to the packet's destination X address. Then, it starts to move in the Y dimension until it reaches the destination router. Since connections in the network are unidirectional, the packets can only travel in the direction of increasing addresses, if necessary wrapping at the edge of the network.

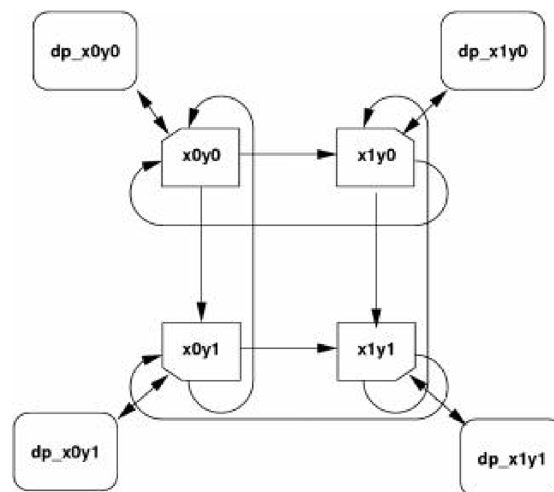


Figure 2.4: Four routers (xYyY), four miniMipses (dp_xXyY) and the data lines that connect them

2.3 C libraries for the *MiniNoC*

There are two C libraries that help create and debug C code for the *mMIPS NoC*: *stdcomm* and *mtools*. The *stdcomm* library implements a message passing communication protocol and *mtools* supports simple debugging using a *printf()* variant.

2.3.1 C message passing library (*stdcomm*)

The *stdcomm.h* and *stdcomm.c* files contain the interface and implementation of a simple message passing library for the networked *mMIPS*. The network interface hardware supports memory mapped, packet based communication. Message passing was chosen because it integrates very well with this type of hardware communication, resulting the smallest possible development and performance overhead costs. The *sc_send()* and *sc_receive()* primitives are used in the C code of the applications running on the *MiniNoC* and they are interpreted by the compiler giving the proper assembly code to the *mMIPS* to access the NI.

2.3.1.1 Send primitive

The *sc_send()* primitive is used for sending data through the network. It has three parameters: relative destination address, a pointer to the buffer where the information is stored, and the number of bytes to be sent:

```
int sc_send(const int address, const void *data, const int size_in_bytes);
```

The *sc_send()* primitive splits the message into 32-bit word data and uses another primitive called *sc_send_word()* to send each one. The *sc_send_word()* primitive before sending each word, first checks if the network interface is ready to accept it.

2.3.1.2 Receive primitive

The *sc_receive()* primitive is used for receiving data. In this case only two parameters are necessary: a pointer to the buffer where the data should be stored and the size in bytes to receive:

```
int sc_receive( const void *data, const int size_in_bytes);
```

The *sc_receive()* primitive receives all the message. It rebuilds it by using another primitive called *sc_receive_word()* which receives each 32-bit word fragment of data. The *sc_receive_word()* first checks if the network interface has the word ready to be read and then it reads it.

2.3.2 *mMIPS* mtools library (mtools)

The *mMIPS mtools* library enables simple debugging and implements functions that are specific to the *mMIPS*. The *mtools* library consists of the files *mtools.h, mtools.c* and requires the sprintf.h, sprintf.c and stdarg_mm.h. The sprintf files and stdarg_mm.h are required for the function *mprintf()*. They contain *svprintf()*, which is equivalent to the standard function *vprintf()* with the difference that the former outputs to a memory location and the latter to stdout.

2.3.3 LCC C Compiler

LCC is a retargetable C compiler. The "target" of a C compiler is the processor for which it generates assembly instructions. The lcc compiler used in this project is version 4.1 and has been ported to the *mMIPS*. A separate assembler (included with lcc) converts the *mMIPS* assembly to machine code that can be uploaded to the FPGA or fed to the SystemC hardware simulator for the *mMIPS NOC*. The script *dolcc* invokes the lcc compiler. As with any other compiler, all source code files must be first compiled and then the resulting object files linked into a binary.

2.4 Running an application

This section explains how an application is run on the *MiniNoC*, and presents different ways of obtaining the timing information once an application is executed. These are methods which will often be used to obtain the metrics of the RTL specification.

When an application is to be run on the *MiniNoC* platform, the first step is to compile the application and the *stdcomm* library with LCC C compiler. After that, run both of them on the SystemC simulator for the *mMIPS NOC*. The *MiniNoC* is ready to be used by just compiling all *mMIPS NOC* SystemC descriptions with the GCC C++ compiler. See *Figure 2.5*.

When the execution has finished some extra timing information can be obtained by means of the output file that gathered the standard output of the execution. Within this file you can observe the steps followed by a word when it is sent from one *mMIPS* processor through all the modules that implement the *mNoC* until it reaches the destination *mMIPS*.

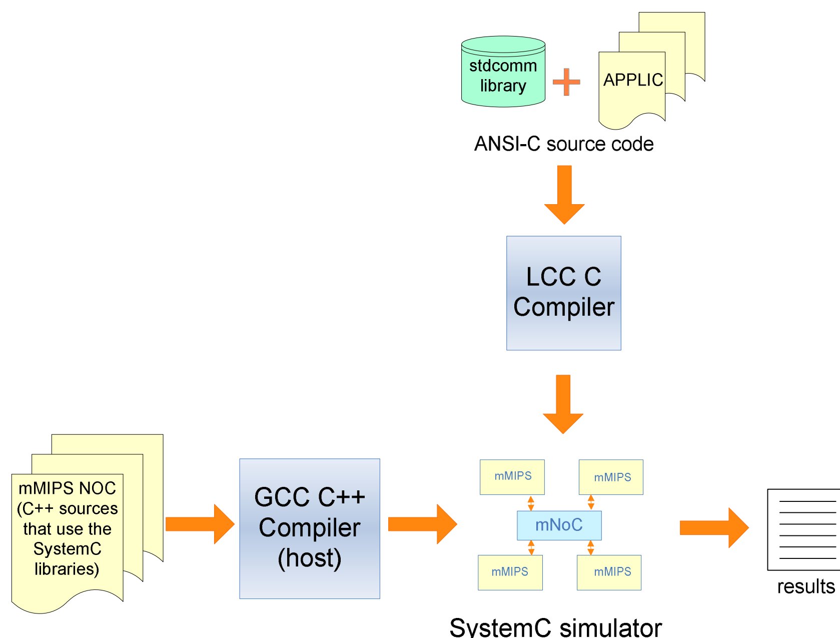


Figure 2.5: Process followed to run an application in the *MiniNoC* platform

The *mMIPS NOC* sources provides the way to find out the last simulation time of the system by means of the *main_net.cpp* file (see *Figure 2.6*). The file incorporates the SystemC *sc_time_stamp()* function (see line 16), for which the outcome can be read in the standard output when the execution has finished. It also includes *sc_time_stamp()* functions for each node of the system, so it is possible to know when each node of the *MiniNoC* has finished by looking at the output file (see line 13). The clock system is also implemented in the *main_net.cpp* file (see lines 5 and 10) with a period of 10. Since the default time unit is 1ns, the clock system period is 10ns.

```
1   //unnecessary lines omitted
2   #include <time.h>
3   …
4   sc_signal<bool> clk;
5   unsigned sim_time = 0, period = 10;
6   int sc_main(int argc, char *argv[]){
7   …
8   while( max_time < 0 || sim_time < (unsigned)max_time ){
9     clk = 0;
10    sc_cycle(period/2);
11    …
12   if( !e01 && dp_x0y1_pc.read().to_uint() == 0x28 ) {
13      cout << "FINISHED x0y1 @ " << sc_time_stamp() << endl;
14      e01 = true; }
15   if( e00 && e10 && e01 && e11 ) {
16      cout << "ALL FINISHED @ " << sc_time_stamp() << endl;
17      break; …
```

Figure 2.6: Extract of the *main_net.cpp* file

At the end of the simulation *mips.vcd* file is produced, which contains a trace of all signals inside the system during the simulation. With this file it is possible to access information about the timing delays through all the modules of the *MiniNoC* platform. It can be viewed by using the *GTKWave* waveform viewer. This visualization tool generates waveforms allowing the examination of the results stored in the trace files after the simulation has been executed (*Figure 2.7*).

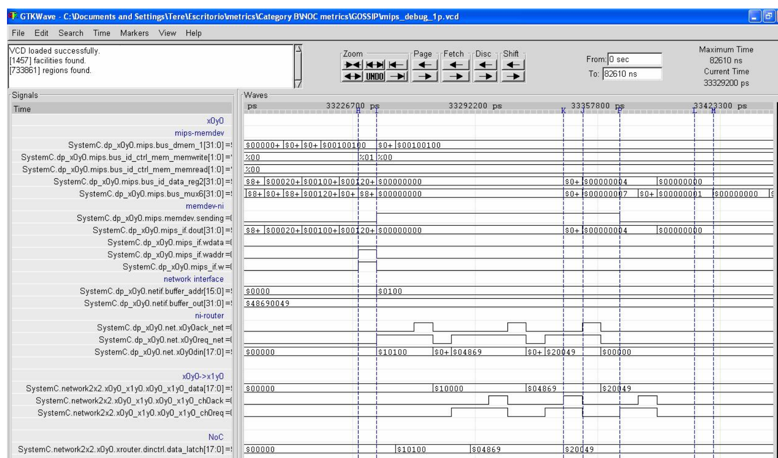


Figure 2.7: View of the VCD file with the waveform viewer

Chapter 3

CASSE tool

3.1 Introduction

CASSE is a system-level modelling and simulation environment [18]. It provides a unified environment that covers system design from parallel application specification to system implementation; by making use of application modeling, architecture modeling, mapping, analysis and progressive refinement.

It aims to ease and speed up the modeling and analysis of *MPSoCs* because it allows initiation of modeling by automatic construction with modeling patterns which are created and configured by means of specification files.

This chapter presents a brief overview of the CASSE tool. It only describes general features and the different stages followed to develop a model with it. However, more detailed information about the tool can be found at [18-21]. In this Master Thesis the CASSE tool will be applied in order to create a new system that will be explained in detail in *Chapter 7*.

3.2 Design methodology

CASSE follows a typical Y-chart methodology [6] where application functionality and architecture are independently modeled and combined in a separate mapping phase.

An important feature of CASSE is that the user controls all stages of the design flow by textual description files. These description files are read and parsed by the tool during elaboration time in order to create and properly configure the desired system model. Since they are easy to modify,creating a new system, the tool simplifies the exploration of several scenarios. Changing these description files does not require recompiling of the existing models.

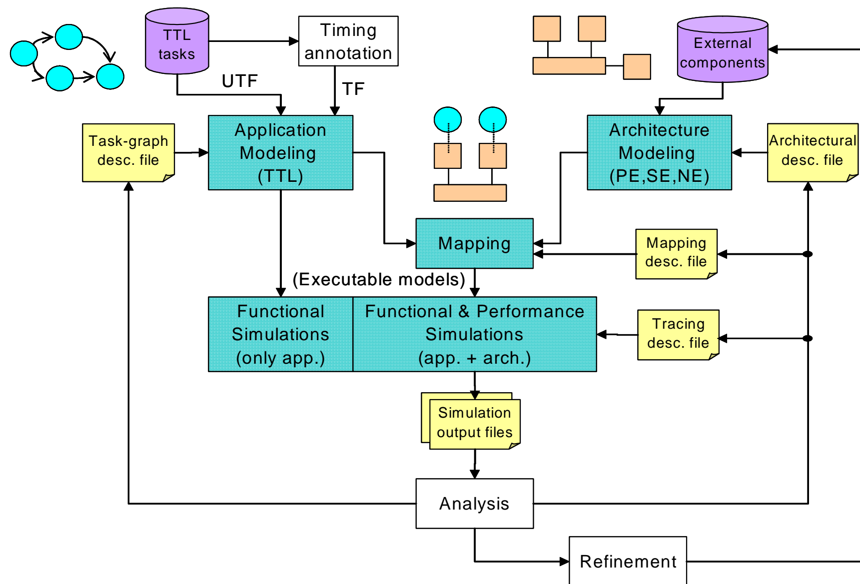


Figure 3.1: CASSE design flow

3.2.1 Application modeling

CASSE applies a parallel programming model based on the Task Transaction Level (TTL) interface [22]. The TTL specification describes the application as a process network where parallel tasks communicate with each other by means of unidirectional channels.

In the tool, tasks containing the application functionality are written in C/C++ (i.e. the functionality per task is fixed at compile time). However, the task graph structure (i.e. port to channel connections) and its configuration (e.g. channel size) are described in a separate text file. The tool uses this to instantiate and bind together tasks and channels creating an architecture-independent executable model of the application.

3.2.2 Architectural modeling

CASSE provides an easy and fast model of the architectural modeling by describing a system as a modular composition of predefined elements (modeling patterns) provided by the tool libraries.

The predefined elements provided are:

- **Processing elements** (PE): model the generic computational units existing on a system, emulating their functionality and timing. An arbitrary number of tasks can be assigned (mapped) to a single PE, but only one task can be active at given time. It provides an operating system functionality with a task scheduler module with multiple schemes and an interrupt controller.

- **Storage elements** (SE): model generic random access elements, such as register files or static RAM memories. It allows the emulation of the behavior of single, dual or multi-port memories existing in the system architecture

- **Network elements** (NE): model shared bus interconnections, such as on-chip shared busses. The principal functionality is to interconnect the processing and storage elements.

All of these elements are highly configurable and it is possible to connect an unlimited number of them in a plug and play fashion by means of the ICCP (Inter-Component Communication Protocol) interface. ICCP is an abstract communication protocol, which defines a point-point interface and a group of communication primitives between two entities named *Inititator* and *Target.* Both the ICCP interface and the library of predefined elements have been developed using the IEEE 1666 SystemC [23] and the OSCI TLM standards [15]. Besides these predefined elements the architecture models can be extended with new functionality by means of **external components** (EC). These ECs can be described at any abstraction level using SystemC and they can be seamlessly added to the architectural models as long as their interfaces are ICCP compliant.

A separate description file is used in order to specify the architectural composition of the system (i.e. number of elements of each type, number of interfaces per element, and their interconnection), and its configuration (e.g. memory map, memory sizes, communication latencies per interface, task scheduler policy, etcetera).

3.2.3 Mapping and execution

An important contribution of the CASSE tool is the straightforward mapping support of the applications onto the architectural models. This feature eases the exploration of different partitioning alternatives with a minimum effort.

The tasks are mapped to the processing elements and the channels to the storage elements. Multiple tasks can be mapped in a single processing element which provides an operating system functionality with a task scheduler module with multiple schemes. The outcome of the mapping stage is an executable model containing the selected application/architecture instance.

CASSE can be used to perform functional simulations of only the application model or performance simulations of the application mapped and executed on the architectural model.

3.2.4 Analysis

During the performance simulations the tool can obtain and record information about the system execution. This allows the user to analyze architectural bottlenecks and system optimizations, so that bottlenecks can be avoided and performance constraints met. Based on this, further iterations might be carried out where both application and the architecture models might be tuned, or new mapping selected.

3.2.5 Refinement

Once the system is ready for implementation the hardware modules can be progressively refined from more abstract to more accurate descriptions in SystemC, and verified within the architectural model just by replacing predefined elements of the tool libraries with external components containing the accurate model.

Chapter 4

CTAP tool

4.1 Introduction

Early in the design flow, an analysis of the TLM model will help to make decisions about the RTL design. The problem is that an untimed model does not contain the timing information necessary to perform this analysis. Therefore, it is necessary to enrich the untimed model with timing information.

This chapter explains the CTAP tool that can be used to provide this timing information. In this Master Thesis the tool will be applied in order to annotate these timing delays into an untimed CASSE (Mixed-Level) model.

4.2 Functionality

CTAP is a tool that provides the number of clock cycles that a C statement requires once it is compiled. The tool provides this information by finding out the number of assembler instructions needed to execute this C statement. CTAP by default assumes that each assembler instruction takes one clock cycle. However, the user can, if required, specify a different duration for each instruction. This can be done for all the assembler instructions in a global way or by making a table that establishes the number of clock cycles for each type of instruction.

The tool provides a configuration file to establish the files that need to be applied; to specify the compiler used; and to configure the number of clock cycles assigned to each assembler instruction. An important feature to mention is that the tool only works with C source code.

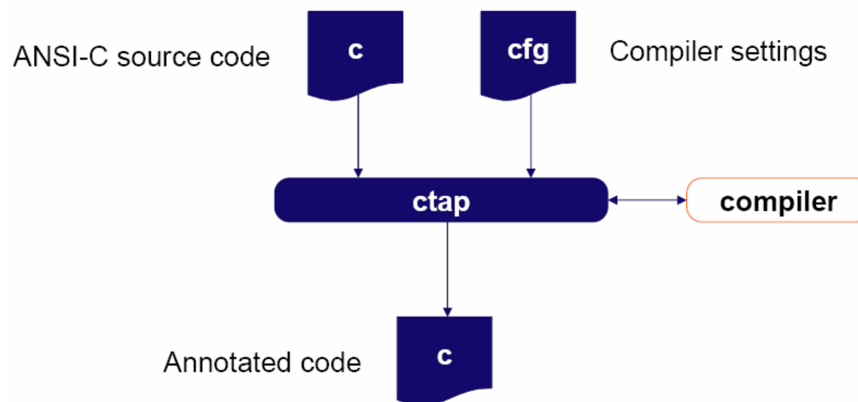
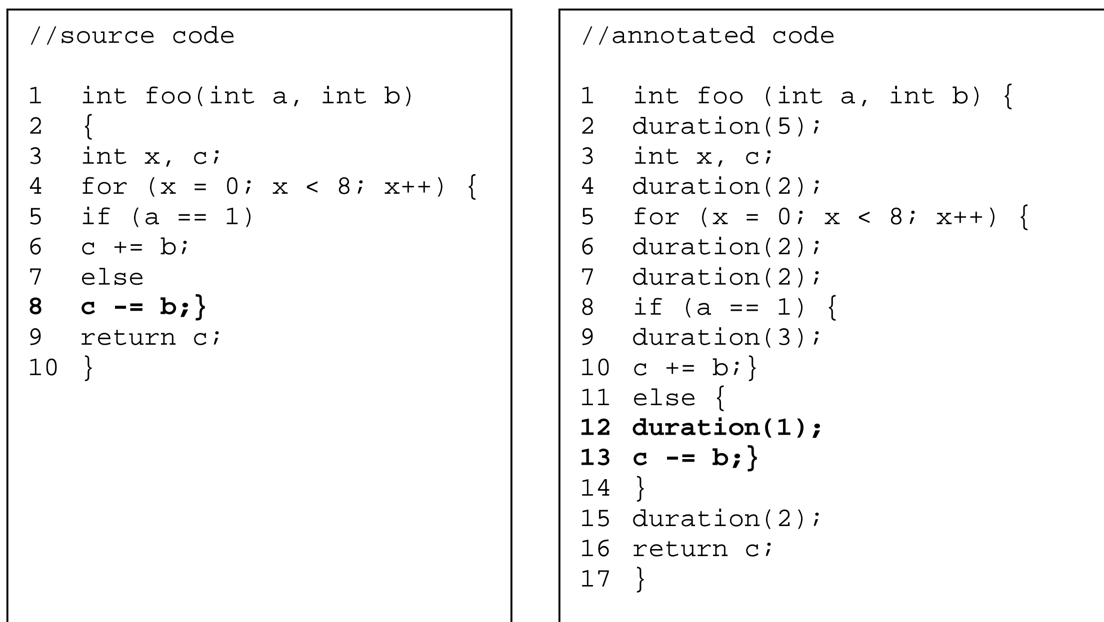


Figure 4.1: CTAP tool process

Once a C file is written and the compiler settings configured, CTAP can be applied. The tool uses the compiler information to annotate the source code with the duration information. This annotation is performed at each C statement indicating the duration information by means of *duration(X)* statements. The *X* number is the number of clock cycles needed to perform that C statement.

A simple example is shown in *Figure 4.2*. Observe that the annotated code is based in the original source code and that the time estimated for each line of code is added by *duration()* functions. The tool has been configured to set each assembly instruction as one clock cycle.



```
//source code

1   int foo(int a, int b)
2   {
3   int x, c;
4   for (x = 0; x < 8; x++) {
5   if (a == 1)
6   c += b;
7   else
8   c -= b;}
9   return c;
10  }
```

```
//annotated code

1   int foo (int a, int b) {
2   duration(5);
3   int x, c;
4   duration(2);
5   for (x = 0; x < 8; x++) {
6   duration(2);
7   duration(2);
8   if (a == 1) {
9   duration(3);
10  c += b;}
11  else {
12  duration(1);
13  c -= b;}
14  }
15  duration(2);
16  return c;
17  }
```

Figure 4.2: Example of timing annotation applying CTAP

Chapter 5

Applications

5.1 Introduction

This chapter explains the application used throughout this Master Thesis: the JPEG decoder. This application has been chosen to performing the comparison between the *MiniNoC* platform and the CASSE Mixed-Level model.

5.2 JPEG application

The decoder takes a JPEG image and converts it to an (uncompressed) bitmap using a decoding method called the baseline decoding process [24]. The multiprocessor decoder is based on a single-processor decoder, and its functionality is presented in *Figure 5.1* that shows the process steps (blue) of the JPEG decoding process.

The decoder takes the compressed image data as its input. It then subsequently applies a variable length decoding [VLD], zigzag scan [ZZ], dequantization [DQ], inverse discrete cosine transform [IDCT], a color conversion and reordering to it. It then obtains the reconstructed image.

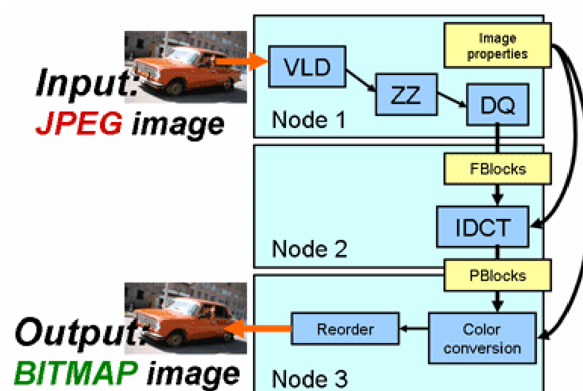


Figure 5.1: JPEG decoding process

The JPEG process has been divided into three tasks which are mapped into three nodes, as depicted in the figure. Each step runs on a separate node of the *mMIPS* network: node1 is at (X, Y) = (0, 0), node2 is at (1, 0), node3 is at (0, 1) and the node (1, 1) remains unused.

An image can be separated into a number of blocks which are grouped in macro blocks –MCU [Minimal Code Unit] - which describe a certain region of the image. The process of creating block and grouping them into MCUs is performed by the encoder. The decoder finds these MCUs one after another in the compressed image data. The decoder can then split an MCU into its representational blocks and decode those. The yellow blocks (Image properties, FBlocks, PBlocks) depicted in *Figure 5.1* describe the data that is sent between nodes (*mMIPS* processors).

The input image applied to the decoder is split at node 1 into macro blocks, which are sent by means of blocks through the *mNoC*. The blocks are received and processed at node 2 with the IDCT function and then forward to node 3, where they are processed and reordered into a macro block. Thus, the application is sequential. This fact makes it possible to know when the macro block is processed and sent from node 1 and the moment when it is received and processed at node 3.

Chapter 6

MiniNoC Modifications

6.1 Introduction

This chapter describes some extra functionality added to the platform and some changes performed which optimize the simulations. It starts with a description of a new instruction added to the instruction set of the *mMIPS* processor which allows the current time of the application to be shown at a specific line. The remaining changes are focussed on making a better comparison analysis: optimizations of the *stdcomm* library and the JPEG application, which optimizes the simulation of the RTL specification; and the extension of the memory sizes of the *mMIPS* processors, allowing a complete evaluation of the Mixed-level model.

6.2 *TIME_STAMP* instruction

As explained in section 2.4, the *mMIPS NOC* sources provide the way to find out the total simulation time of the system. However, it is not possible to discover the current time in a specific line in the source code of the application. This is because the compiler used in the *mMIPS* processor of the *MiniNoC* is the LCC C compiler whereas the applications loaded in the hardware simulator need to be written in C language, and not in SystemC. Therefore, they obviously are not aware of any function that exists within the SystemC library. To solve this problem it is necessary to add an additional assembly instruction to the *mMIPS* processor to obtain the time stamp at which a specific line in the source code of the application is executed in a module (processor, router…) of the *MiniNoC*.

Providing the original *MiniNoC* with this extra functionality involves not only changes in the *mMIPS* processor but also in the LCC C compiler. The reason is that if an application is to be run in the *MiniNoC*, this C program should be compiled with LCC. The result is a binary file which contains the assembly code that is uploaded into the instruction memory of the *mMIPS*. This involves changes in both the LCC C compiler and the *mMIPS* processor.

The new assembly instruction introduced to the instruction set is named *TIME_STAMP*, which can be placed in any part of the application where it is desired to obtain the current time of the system when is executed. This *TIME_STAMP* instruction is useful since it adds accuracy to the measurements of the metrics performed in the RTL specification.

6.3 New *stdcomm* library

As explained in section 2.2.1, the *mMIPS* processor is a simple *MIPS* version with a reduce instruction set. Compared to the MIPS it has a reduced instruction set which means that some operations need to be done in software, for example the modulo, shift, multiplications and division functions.

The *sc_send()* and *sc_receive()* functions from the original *stdcomm* library include the modulo instruction in several C statements. Since modulo function is one of the instructions that are not among the instruction set, each time the *sc_send()* and *sc_receive()* functions are called a lot of instructions are executed. This means large instruction code size and long run time. Moreover, the place where the modulo function is located is relevant since affects the size of the message. As a result, when the message size increases so does the simulation time required which increases with an almost exponential manner.

A modification of the *stdcomm* library has been performed in order to solve this problem. By just changing the modulo function with a C statement which performs the same functioning but with a operator that is done by hardware.

As a result of this modification the communication primitives have been optimized and the run time considerably reduced. For example, the performance of the JPEG application simulation time was reduced from 8 hours to 48 minutes.

6.4 JPEG decoder

In order to perform the metrics in the JPEG decoder the *verbose* option of the application has been disabled. This option allows the user to observe the steps followed by the decoder during the execution. It is based on the use of *mprintf()* functions (see section 2.3.2) distributed along the application which performance requires a huge number of clock cycles. This number of clock cycles caused by the continued accesses to memory introduces inaccuracies in the comparison. Therefore, this information has been removed since is not relevant for the execution of the application.

6.5 Memory extension

The *MiniNoC* reserves a zone of the RAM memory for the *USER DATA*. In this 3.5kbytes memory space is stored the data used in the application. This memory size does not allow loading and storage of large images for the input and output of the JPEG decoder.

In order to be able to store a larger input image, the map memory of the first node was changed to enlarge the memory user space. In addition, the memories of other nodes were also enlarged to store all required intermediate data and the output image.

Chapter 7

CASSE Mixed-Level model

7.1 Introduction

This chapter presents the CASSE Mixed-Level model and the steps followed in its implementation: application modeling, architectural modeling, mapping and time annotations. Each section of the chapter presents one step of the process.

The application modeling section describes how the applications were modelled into the CASSE Mixed-Level model. Secondly, the architectural modeling describes the different stages followed to perform the Mixed-Level model and what modules it was composed of. Thirdly, follows the mapping section that illustrates how the JPEG application was mapped. Finally, the time annotation section presents how the untimed model was annotated with time.

7.2 Application modeling

In order to model an application into CASSE, the tasks of the application need to be compliant with the task interface of CASSE. According to that, tasks required to be in put into a C++ class and inherit from the class *task_if* of the tool.

Since the *MiniNoC* platform uses message passing to communicate processors, it is necessary to adapt the communication primitives to be able to compliant the protocols of communication of CASSE. As any other application, the communication primitives also require to compliant the task interface of CASSE. Therefore, the library *comm w*hich implements them needs also to inherit all the attributes and operations from the *task_if* of CASSE.

On the other hand, the tasks mapped on the platform call the communication primitives to establish communications with the rest of the nodes. Therefore, the tasks of the application need to inherit the functions (communication primitives) from the *comm* library to be able to use thems.

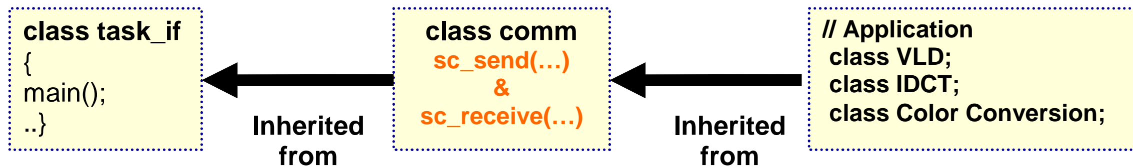


Figure 7.1: Class hierarchy

As a result, *Figure 7.1* illustrates the class hierarchy proposed. The *task_if* class is the base class. The *comm* library is a derived class which inherits all the capabilities from the base class but also adds its own functions *sc_send()* and *sc_receive()*. The classes of the application as a derived class from the *comm* class, inherit the capabilities of the base class and also from the *comm* class. This fact allows the application to be compliant with the task interface and to access, to send and receive functions.

Note that the base class is unchanged by the process. The *comm* library only needs to be modified once since it depends on the platform, and can be used by any application once modeled. And finally, note that the application is independent of which platform is been mapped.

7.2.1 *Comm* class

The *comm* library is a new library that implements the communication primitives of the Mixed-Level model: the *sc_send()* and *sc_receive()* functions. These functions are based on the original *MiniNoC* functions implemented in the *stdcomm* library, which by means of loads and stores to memory locations control the communications within the network.

Since the Mixed-Level model communicates with the network at a transaction level, the communication primitives have been slightly modified in order to adapt them to the ICCP protocol.

As explained in section 4.2, ICCP is an abstract device level communication protocol, which defines a point-to-point TLM interface and a group of communication primitives between two entities named *Initiator* and *Target*. Basically, ICCP is used to interconnect all architectural components used in CASSE and allows the communication among them. The ICCP interface provides two basic methods for communication between an *Initiator* and a *Target*: *readBurst()* and *writeBurst().*

In practice, to adapt the communication primitives to the ICCP required a translation of the loads and stores into write and read methods. As a result, the *sc_send()* and *sc_receive()* functions of the *comm* library allow performance of the transactions required to communicate between nodes of the model.

As the original implementation, the *sc_send()* primitive of the *comm* library splits the message into words of 4 bytes and calls the *sc_send_word()* function, which sends each 32-bit word data. As introduced above, in order to adapt them to the ICCP transaction-level communication protocol of CASSE, the *readBurst()* and *writeBurst()*

communication methods have been added at the *sc_send_word()*, that performs the transactions with the network through the *Initiator* entity. See lines 4 and 8 of *Figure 7.2.*

The *sc_receive()* of the comm library is also implemented based on the same functionality of the original *MiniNoC's stdcomm* library. The function collects the packets and rebuilds the message by putting them all together. By means of calls to the *sc_receive_word()* primitive each word is read from the network. The function first checks if the data is available and if that is the case, then reads it. Since the *sc_send_word()* function is where the communications with the network take place, the *readBurst()* and *writeBurst* methods of the ICCP interface are added in order to adapt it to the ICCP protocol.

```
1   int comm::sc_send_word(const int *ctrlword, const int *data,
    int try_count){
2      InitiatorPort* InitP = VML->getInitiatorP(0);
3      …
4      if(ReadBurst(InitP,0x80000004,4,&aux) == ICCP_ERROR){..}
5
6      while(!(aux & SC_BIT_SEND_READY));
7      aux = *data;
8      WriteBurst(InitP,0x80000000,4,&aux);
9      aux = *ctrlword | SC_BIT_SEND;
10     WriteBurst(InitP,0x80000004,4,&aux);
11     …}
```

Figure 7.2: Extract of the *sc_send()* primitive code of the *comm* class

7.2.2 Gossip application

A simple application written in C++, called *Gossip* has been developed to check the correct functioning of the *comm* class and the architecture of the Mixed-Level model.

The application is composed of 4 tasks which send and receive a message through the network by means of calls to the *sc_send()* and *sc_receive()* communication primitives.

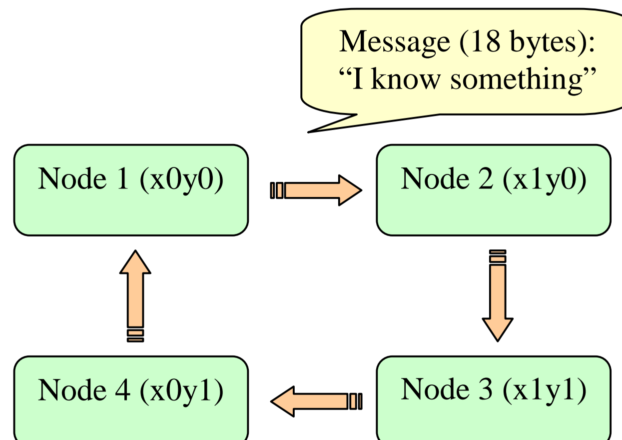


Figure 7.3: Gossip for 2x2 network

Node x0y0 sends the text "*I know something!*" to node x1y0 and then listens for a return message from any node. When it has received that message, it quits. All other nodes wait for an incoming message, forward it to the next node and then exit. In this way, the *Gossip* application tests if the whole system has been set up correctly.

The four tasks are created in a separate description file named *taskgraph.txt*. This file is used in the tool to describe the task graph structure and its configuration.

7.2.3 JPEG modifications

The JPEG decoder has already been explained at *Chapter 5*. This section briefly explains the modifications done to the application in order to adapt it into CASSE.

Since the JPEG application is described in C, the three tasks were first moved into C++ classes. In order to model the application into CASSE, as explained above, the three classes were defined as a derived class of the *comm* library.

Finally, it was necessary to remove some code from the application which was specific from the *MiniNoC* platform. Such as calls to specific functions of the *mMIPS* processor, as the *mt_halt()*and *mprintf()* functions which enables simple debugging implemented at the *mMIPS mtools* library.

Once the modifications where performed, the three tasks were created by means of the *taskgraph.txt* description file (*Figure 7.4*).

```
1   # Process Network file #
2   # Tasks creation #
3   .CREATE -TASK step1 -N_PORT 0 ;
4   .CREATE -TASK step2 -N_PORT 0 ;
5   .CREATE -TASK step3 -N_PORT 0 ;
6   # eof #
```

Figure 7.4: Application description file

Note that a number of ports must to be assigned to each task. This assignment is required because CASSE applies a parallel programming model based on the TTL interface. The TTL specification describes the application as a process network where parallel tasks communicate with each other by means of unidirectional channels. Since the Mixed-Level model communicates by message passing, the tasks do not need to communicate among themselves by means of channels, so no ports are needed. However, the description file of the tool requires adding those commands to be able to create the tasks correctly.

7.3 Architecture modeling

The Mixed-Level model is composed of two levels of abstraction: TLM (Transaction Level Modeling) and RTL (Register Transfer Level). The processor tiles of the model are at a TLM level, and the NOC component is at a RTL level since it includes the *mNoC* network of the *MiniNoC* platform (*Figure 7.5*).

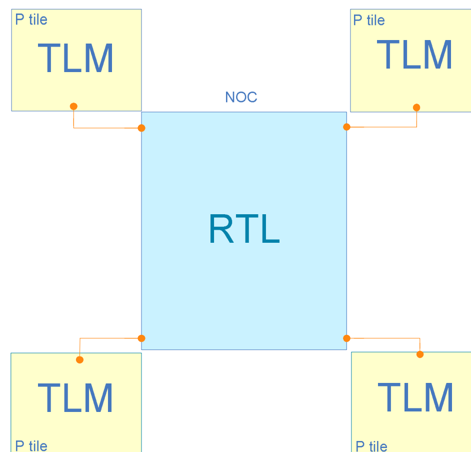


Figure 7.5: Mixed-Level model

This section describes each of the steps followed to obtain the model. It will start by describing how the processor tiles were created and then, how the *mNoC* network was integrated and adapted to the model.

7.3.1 Creating the 4-tiles platform with CASSE

The aim of this first step is to create the four processor tiles of the new model with CASSE. In order to achieve this step, the four *mMIPS* processors tiles were modelled with CASSE predefined components which are provided by the libraries of the tool. The *mMIPS* processors were modeled with Processing Elements (PE), the local memory with Storage Elements (SE) and the *MEMDEV* module with Network Elements (NE). See *Figure 7.6*.

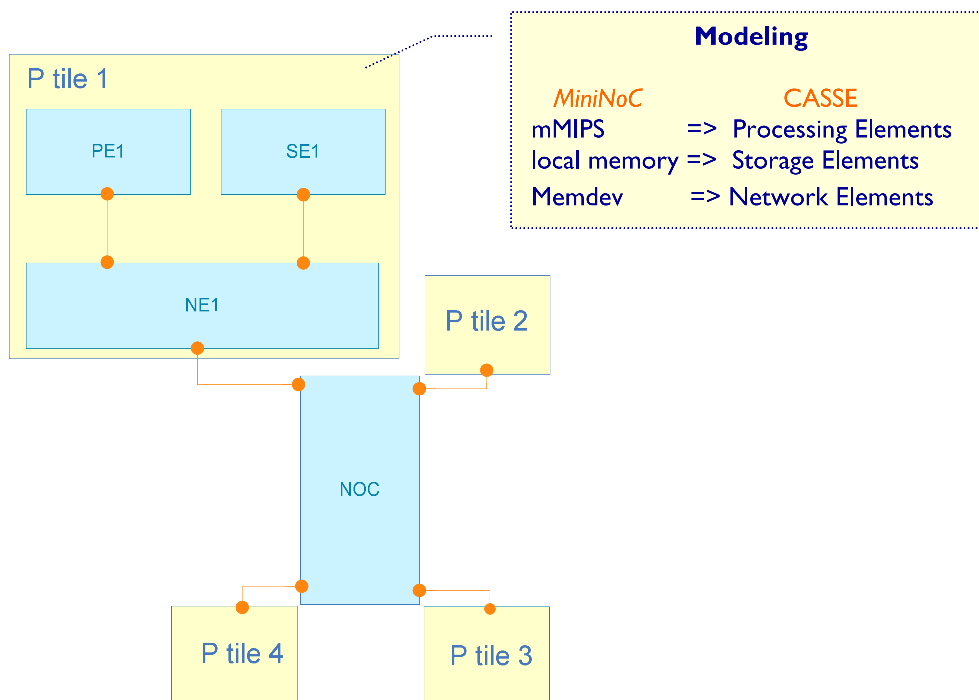


Figure 7.6: Modeling the *MiniNoC*'s PTile with CASSE

7.3.2 Adapting and integrating the *mNoC* component

Once the processor tiles were replaced with the predefined elements of CASSE and the functionality of the platform was checked, the next step was to adapt and integrate *mNoC* components of the *MiniNoC* into the initial model.

In order to integrate the *mNoC,* an External Component (EC) of CASSE named NOC was introduced. It is composed of the four routers and network interfaces of the original *MiniNoC* whose sources are described in RTL.

The next step was to connect the NOC module with the P tiles. Since the *MiniNoC* is described at a low level (RTL) and does not comply with the ICCP protocol, an adaptor module was essential to carry out the transactions.

The adaptors are added to change the levels of abstraction by converting the timing-accurate signals of the network interface to a transaction view of the bus. This feature allows parts of the design to be simulated at the transaction level while other parts of the design can be simulated on the detailed hardware level. See *Figure 7.7*.

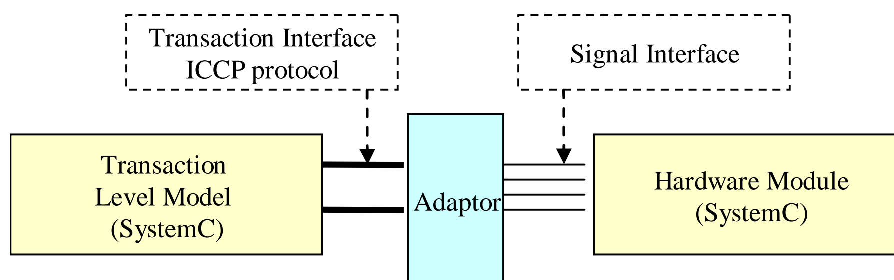


Figure 7.7: Adaptors connect abstract transaction-level models to signal-level modules

To sum up, the adaptors were added to the NOC external component in order to adapt the components of the *MiniNoC* to the ICCP interface, and as a result the NOC is able to perform transactions with the P tiles.

7.3.2.1 Adaptor

The adaptor is implemented with a *Target* port where the transactions are made by means of the ICCP protocol. Bound to the Target port is a slave, which implements the functions *read()* and *write()* of the ICCP interface which carry out the transactions.

Also a thread is described in this module, named *exec().* This thread accesses to the signals of the module and carries out with the handshake protocol of the *MiniNoC*'s Network Interface module. For that some form of communication between the thread and the slave is needed to perform the transactions. This communication was implemented by buffers which store the data that needs to be read and to be written.

Figure 7.8 shows the implementation of the adaptor and one of the processor tiles of the model. When the *sc_send()* or/and *sc_receive()* primitives are called into a task mapped at a PE, a transaction starts with a call to the *writeburst()/readburst()* methods of the ICCP interface. These transactions are carried out in the adaptor module

by calling the interface write/read functions which are implemented in the slave. The slave communicates with the thread which manages the appropriate signals according to the transaction requested.

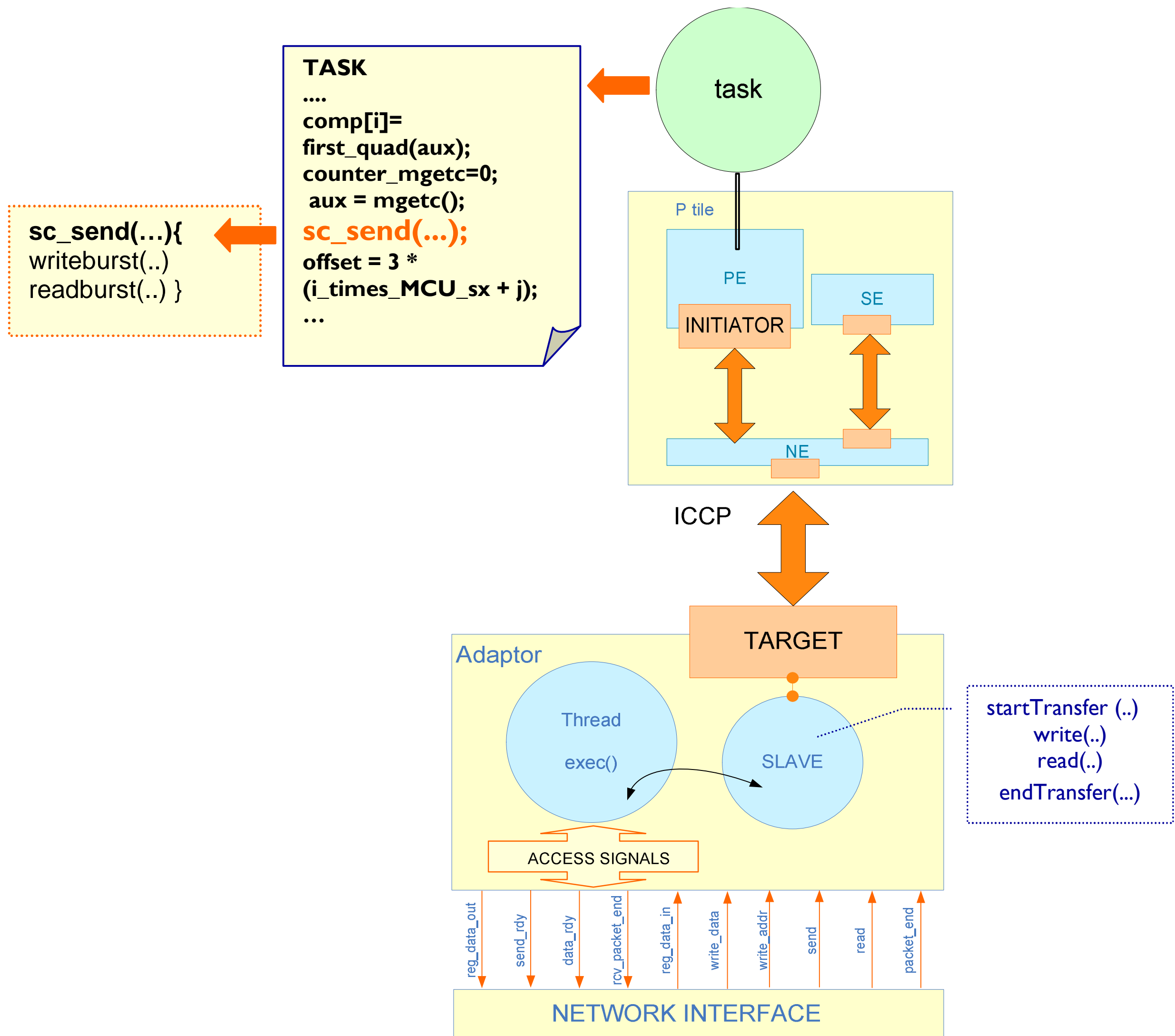


Figure 7.8: Adaptor Implementation

7.3.2.2 NOC component

As explained above, the NOC external component contains the original *mNoC* network of the *MiniNoC*: the four routers and the four network interfaces. The integration of these *MiniNoC* components was achieved by means of a file called *NINET.h,* (Network Interfaces and NETwork). It includes four instances of the network interface and one of the *network2x2.h,* which already includes the interconnection of the four routers (*Figure 7.9*). Finally, the NOC module was completed by connecting the *NINET.h* file to the 4 adaptors. See *Figure 7.10*.

```
1    // ninet.h file
2    #include <systemc.h>
3    #include "network2x2.h"
4    SC_MODULE(NINET){
5    // PORTS
6       sc_in< bool > clk;
7       sc_in< bool > rst;
8       sc_in< sc_bv<32> > nix0y0_reg_data_in;
9       sc_in< bool > nix0y0_write_data;
10      …
11   // Pointer declarations
12      NETWORK_INTERFACE *nix0y0;
13      NETWORK_INTERFACE *nix0y1;
14      …
15      NETWORK2x2 *net;
16      SC_CTOR(NINET){
17      //Objects instanciated
18         nix0y0 = new NETWORK_INTERFACE("nix0y0");
19         nix0y1 = new NETWORK_INTERFACE("nix0y1");
20         …
21         net = new NETWORK2x2("net");
22      // Mapping signals to the objects ports
23         net->clk(clk);
24         net->rst(rst);
25         net->x0y0dout(nix0y0_data_in);
26         net->x0y0req_dp(nix0y0_req_in);
27   …}};
```

Figure 7.9: Extract of the *Ninet.h* file

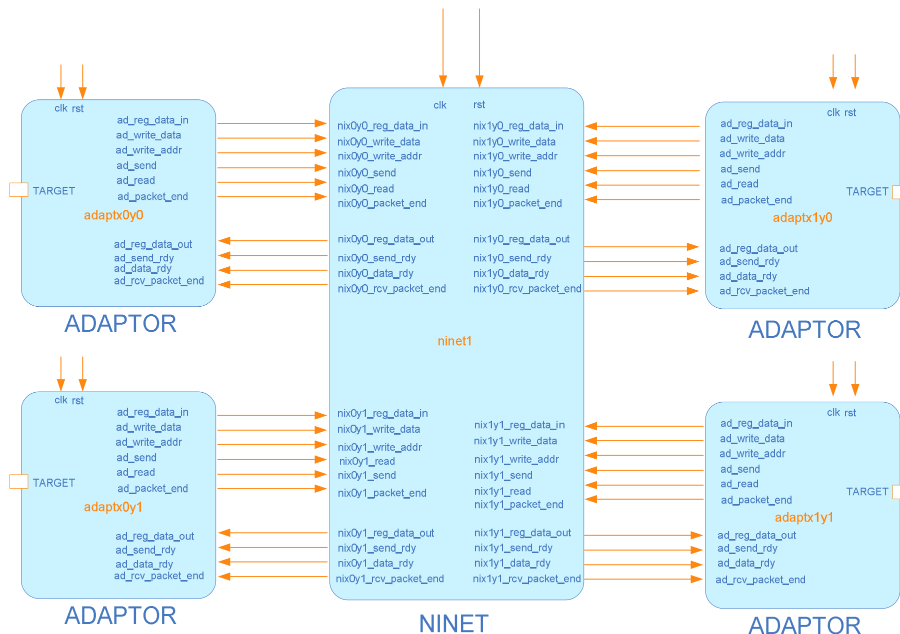


Figure 7.10: NOC connections

7.3.3 CASSE Mixed-Level model

Once the NOC component was added to the initial model performed with the four processor tiles, the model was completed. As a result, the architecture of the

platform is composed of four P tiles - that performed the functionality of the P tiles of the *MiniNoC* platform- and a *NOC* element -which adapts and integrates the *mNoC* component in *CASSE*.

Each processor tile is performed with *CASSE*'s predefined elements: PE, SE and NE. The *NOC* component contains the routers, network interfaces (*mNoC* components), and the adaptors added to adapt them to the ICCP protocol.

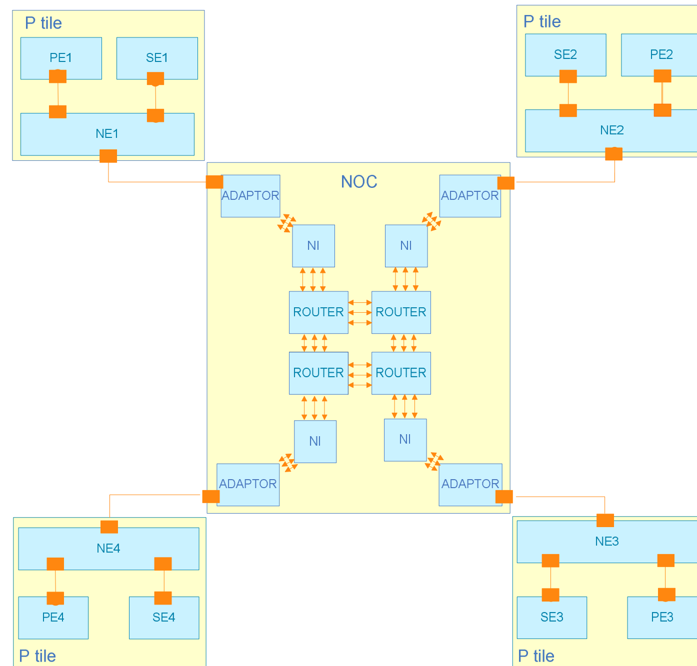


Figure 7.11: CASSE Mixed-Level model

Each of the modules of the model was created and configured by use of a textual description file called "architecture file". This description file is used by the tool in order to specify the architectural composition of the system. To specify the whole system around 100 lines of code were necessary. Each processor tile required approximately 25 lines, where the predefined and external elements were created and configured. Each element (processing, storage, network and external) is created by indicating the number of interfaces and interconnections. It is also configured by specifying the memory map, memory sizes, communication latencies per interface, task scheduler policy and so on. The clock also was added here, with the same 10 ns clock period as the *MiniNoC* platform. The lines used to create and configure Tile 1 are shown in *Figure 7.12*.

```
1   # architectural file #
2   .CREATE -CLOCK clock1 -PERIOD 10 -UNIT SC_NS ;
3   .CREATE -EXTERNAL noc ;
4   # TILE 1 #
5   .CREATE   -PROCESSING proc1 -N_INIT 1 ;
6   .CONFIGURE      -PROCESSING proc1 -INIT 0 -WIDTH 32 -LAT 0 0 0 0 -
    CONNID 0 ;
7   .CONFIGURE      -PROCESSING proc1 -TS MLQ 100 50 0 ;
8   .CONFIGURE      -PROCESSING proc1 -PTW YIELD 5 2 1 5 ;
9   .CONFIGURE      -PROCESSING proc1 -INIT 0 -MEMORYMAP 0x00000000
    0x000fffff mem1 ;
10  .CREATE   -STORAGE     mem1 -N_TARGET 1 ;
11  .CONFIGURE      -STORAGE     mem1 -SIZE 1048576 ;
12  .CONFIGURE      -STORAGE     mem1 -TARGET 0 -WIDTH 32 -LAT 0 0 0 ;
13  .CREATE   -NETWORK     bus1 -N_INPUT 1 -N_OUTPUT 2 ;
14  .CONFIGURE      -NETWORK     bus1 -ARBITER ROUNDROBIN ;
15  .CONFIGURE      -NETWORK     bus1 -WIDTH 32 -BUFFERED n -I_LAT 0 0 0 -
    O_LAT 0 0 0 0 ;
16  .CONFIGURE      -NETWORK     bus1 -OUTPUT   0 -RANGE 0x00000000
    0x000fffff ;
17  .CONFIGURE      -NETWORK     bus1 -OUTPUT   1 -RANGE 0x80000000
    0xffffffff ;
18  .CREATE -LINK LPB1 -WIDTH 32 ;
19  .CREATE -LINK LBM1 -WIDTH 32 ;
20  .CREATE -LINK LECB1 -WIDTH 32 ;
21  .BIND -CLOCK clock1 TO -PROCESSING proc1 ;
22  .BIND -CLOCK clock1 TO -STORAGE mem1 -TARGET 0 ;
23  .BIND -CLOCK clock1 TO -NETWORK bus1 -INPUT ;
24  .BIND -CLOCK clock1 TO -NETWORK bus1 -OUTPUT ;
25  .BIND -LINK LPB1 TO -PROCESSING proc1 -INIT 0 ;
26  .BIND -LINK LPB1 TO -NETWORK bus1 -INPUT 0 ;
27  .BIND -LINK LBM1 TO -STORAGE mem1 -TARGET 0 ;
28  .BIND -LINK LBM1 TO -NETWORK bus1 -OUTPUT 0 ;
29  .BIND -LINK LECB1 TO -NETWORK bus1 -OUTPUT 1 ;
30  .BIND -LINK LECB1 TO -EXTERNAL noc -TARGET 0 ;
31
32  // configuration lines of tiles 2, 3, 4 omitted
```

Figure 7.12: Architecture description file

7.4 Mapping

Now that the platform is set up, the next step is to map the tasks composing the JPEG application on the architecture model.

As explained at section 5.2, the JPEG decoder is divided into three tasks. Each step runs on a separate node: node1 is at (X, Y) = (0, 0), node2 is at (1, 0), node3 is at (0, 1) and the node (1, 1) remains unused.

Since CASSE already provides direct support for application to architecture mapping, this procedure is rapidly described by means of the mapping description file. The JPEG task mapping only required 3 lines of code, where each line maps one task to its respective processor (*Figure7.13*).

```
1   # mapping file #
2   .MAP -TASK step1 INTO -PROCESSING proc1 ;    # x0y0 #
3   .MAP -TASK step2  INTO -PROCESSING proc2 ;   # x1y0 #
4   .MAP -TASK step3  INTO -PROCESSING proc4 ;   # x0y1 #
5   # eof #
```

Figure 7.13: Mapping description file

7.5 Time annotations and analysis

The timed model is a modeling approach where timing delays are inserted into an untimed model. These annotated delays are the timing information of the micro-architecture level, which are performed by means of simple *wait(sc_time)* statements related to the computation time of a specific functionality. This timing information will be obtained from the RTL specification.

7.5.1 Alternatives for timing annotation

In order to perform the comparison, it is necessary to annotate the *CASSE* untimed model with timing information obtained from the RTL specification. To achieve this process, different alternatives have been proposed:

<u>Option 1:</u>

An estimation of the execution time is made by means of instrumenting each C statement of only the application to determine the number of assembly instructions generated by the compiler for that statement. The resulting execution time in clock cycles is then obtained by multiplying this number of assembly instructions with an estimate of the average number of clock cycles per assembly instruction (in the range of about 1.0 to 1.2 or so). The result can then be used again to annotate each C statement of the application in the CASSE model. The advantage of this approach is that it only requires compiling the application (but not executing it). Besides losing some accuracy because of ignoring caching effects etc, the execution times used in the model are estimations based on the average number of clock cycles per assembly instruction which ignores variations due to more or less usage of more or less expensive (in terms of clock cycles) instructions.

<u>Option 2:</u>

An estimation is made of the execution time of blocks of C code in the application. We obtain the number of assembly instructions generated by the compiler for each C statement by multiplying the number of assembler instructions determined for a C statement of the code with an estimate of the average number of clock cycles per assembly instruction (similar to option 1). With the timing information for each C statement it is possible to obtain the number of clock cycles for each block of C code in the application by adding counters that measure each one. The resulting measures can be annotated in the CASSE model. As in option 1, the advantage is that it only requires compiling the application.

<u>Option 3:</u>

This option proposes a way of reducing inaccuracies in option 2, due to the estimation of the global number of clock cycles for each assembler instruction. The idea is to analyze the number of clock cycles needed for each assembler instruction in different parts of the code and make an average. With a table containing the average of number of clock cycles of each assembler instruction we are able to configure the tool that provides the timing information. As a result, we have a better estimate of the number of clock cycles that each C statement takes in its application.

Option 4:

Instrument the RTL specification to measure the exact number of clock cycles taken by each C statement in the application. This approach involves both compilation and execution of the total RTL specification (application + *MiniNoC* platform) to obtain these values. Then use the resulting information to annotate each C statement of the application in the CASSE model with that number of clock cycles.

Option 5:

Instrument the RTL specification to measure the exact number of clock cycles that large parts of the C code in the application takes. Use these estimate to annotate the C code of the application in the CASSE model. This approach has the same disadvantage as option 4 since it requires compilation and execution of the total RTL specification and the estimation of larger parts of the C code might be rather inaccurate.

Note the difference in level of granularity at which the different approaches work (C statement for options 1, 4 and C block for options 2, 3 and 5)

Of the five options proposed above we decided to explore option 2 and option 5. The first is the most likely option in the context of a real design and also the most interesting one from a design/tool flow point of view. Option 5 has been chose because it provides a way of considering the instrumentation RTL option without spending much effort on it. Finally, we have not considered options 3 and 4 since they require spending a lot of time and the accuracy added to the model is questionable.

An evaluation will be performed by analyzing the accuracy of the two options in order to finally decide the best option. To address that evaluation the measurements for both options have been taken for the JPEG decoding the 32x24 pixel surfer image. And the CTAP tool has been chosen for the estimation of the timing annotation for option 2.

Figure 7.14: Input image *surfer.jpg*

7.5.2 Timing annotations in CASSE

CASSE provides two functions for annotating the timing information: *DELAY_CYCLES(ncycles)* and *DELAY_TIME(period,time_unit)*, which are basically performed with the *wait(sc_time)* function of the SystemC library. The difference between them is that in the first one it is possible to set the number of clock cycles (the time waiting is derived from the clock connected to the PE where the task executes), while the second one the time can be entered directly by indicating the value and the time unit used.

Both of these methods have been applied to the two options contemplated for the timed Mixed-level model. The *DELAY_TIME()* function was used to annotate the delays at the instrumentation RTL option since timing is annotated in nanoseconds, and

the *DELAY_CYCLES()* function was used for annotating the timing information provided by the CTAP tool which is in cycles.

7.5.3 *mMIPS* Instrumentation

The *mMIPS* instrumentation annotated model consists of instrumenting the RTL specification to measure the exact number of clock cycles of larger parts of the C code. The number of clock cycles can be obtained by using the time stamp functions in the JPEG application. With the resulting information we are able to annotate large parts of C code in the CASSE Mixed-Level model.

To illustrate the process taken to annotate the timing information, we have chosen as an example, the IDCT function of node 2 (see *Figure 7.15*). First of all, time stamp functions were added before and after the IDCT function was called by means of the *TIME_STAMP* instruction. Secondly, the JPEG application was run in the SystemC simulator for the *mMIPS NOC* as explained in section 2.4. When the application ran the timing information was shown in the standard output and the total number of clock cycles taken to perform the IDCT was obtained from the difference between the two time stamps. Then, the average was calculated. Finally, the average obtained was annotated in the model by means of the *DELAY_TIME()* function (see line 7 of *Figure 7.15*). This same process was followed for the three tasks of the application.

```
1  //STEP 2:        IDCT
2   …
3  while(blocks_remain){
4  sc_receive(&input, sizeof(FBlock));
5  IDCT(&input, &output);
6
7  DELAY_TIME(1141570,SC_NS);
8
9  sc_send(ADDR_STEP2TO3, &output, sizeof(PBlock));
10 sc_receive(&blocks_remain, sizeof(int));}
11 …
```

Figure 7.15: Annotation example of the *mMIPS* Instrumentation model

The disadvantage with this annotation is that the resulting annotated model can only be compared for the performance of same image –*surfer.jpg*- in which the timing has been taken. In this way, it is not possible to compare with any other images, and the comparison is limited to a single image. Another disadvantage is that it requires compilation and execution.

7.5.4 Applying the CTAP tool

As explained in section 4.2, CTAP provides the number of clock cycles that a C statement requires once it is compiled. This information is obtained from the compiler that is applied to the tool, and the configuration file in which the number of clock cycles per assembler instruction is specified. Once the tool is applied, the file is annotated with this timing information.

The *mMIPS* LCC C compiler does not suit well with the CTAP tool. The problem with using the LCC in CTAP is that LCC does not support the standard STABS debugging format which makes it impossible to find the relationship between assembler instructions and lines in the C code. Since is not possible to use LCC, the *mMIPS* GCC compiler, which fits with CTAP perfectly, will be applied instead.

The annotated model is obtained by applying the CTAP tool . Firstly, the CTAP tool was applied to the JPEG decoder and to the *comm* library. The compiler used with CTAP was the GCC compiler and the tool was configured with the default option which assumes one clock cycle for each assembler instruction. As a result, the code was annotated with duration information of each C statement. The next step was to annotate that timing information in the JPEG application and the *comm* library of the CASSE untimed model by means of the *DELAY_CYCLES()* function. Sequential parts of the code were annotated by adding counters and the rest was annotated for each C statement.

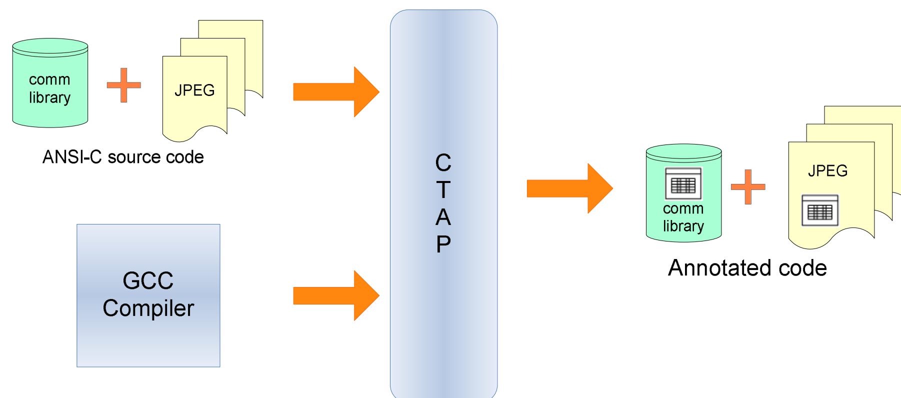


Figure 7.16: Process followed to apply CTAP to the JPEG decoder

An example, the IDCT function has been presented in *Figure 7.17*, in which we are able to observe that timing information has been added to each C statement by using the *DELAY_CYCLES()* function.

```
1   //STEP 2:        IDCT
2   #define CMUL(C, x)  (((C) * (x) + (1 << (C_BITS-1))) >> C_BITS)
3   ..
4   DELAY_CYCLES(k*34);
5   z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
6   DELAY_CYCLES(k*34);
7   z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
8   DELAY_CYCLES(k*3);
9   z3[1] = z2[1];
10  DELAY_CYCLES(k*3);
11  z3[2] = z2[2];
12  ...
```

Figure 7.17: Annotation example of the model

Since the tool was configured with one clock cycle per assembler instruction, no pipeline effects have been considered. In practice, it can be assumed that CPI is 1.3,

which is more realistic than the 1 configured. As a result, it can be assumed a corrector factor of k=1.3.

As explained in section 2.2.1, the *mMIPS* processor has a reduced instruction set which means that some operations need to be done in software, such as multiplications, divisions, modulo, variable distance shifts and all floating point operations. This is the main error introduced by using the *mMIPS* GCC compile. This error is considerable since the compiler does not take account of those operations done by software. Therefore, the assembler instructions provided by the compiler do not match with the real number of assembly instructions that the LCC compiler would required to perform. As a result, CTAP would annotate a much lower number of assembly instructions, introducing an error.

7.5.5 Correcting the compiler error

Owing to the problem that the CTAP tool has with the *mMIPS* LCC C compiler, we tried to correct the error introduced by using the *mMIPS* GCC compiler instead. The solution proposed was to solve the problem with the functions done by software by adding manually the clock cycles required for those C statements. To achieve this it was required to measure them by the instrumentation of the RTL specification. The number of C statements used by these functions is low. However they are used frequently. Therefore, they can be easily identified and measured. Note that this scenario only occurs at the JPEG application and not at the *stdcomm* library where those statements have been previously removed (see section 6.3).

First, the CTAP tool was applied to the JPEG files and *comm* library. Second, the C statements with operations required to be done by software, such as the modulo, shift, multiplications and division functions, were identified. For each C statement identified the time stamps were added before and after by means of the *TIME_STAMP* instruction. After that, the JPEG application was run in the SystemC hardware simulator for the *mMIPS NOC* and the timing information was shown in the standard output.

```
1   //STEP 2:        IDCT
2   …
3   #define CMUL(C, x)  (((C) * (x) + (1 << (C_BITS-1))) >> C_BITS)
4   DELAY_CYCLES(693);
5   z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
6   DELAY_CYCLES(676); // add manually
7   z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
8   DELAY_CYCLES(k*3); // provided by CTAP
9   z3[1] = z2[1];
10  DELAY_CYCLES(k*3);
11  z3[2] = z2[2]; ....
```

Figure 7.18: Annotation example of the modified model

The total number of clock cycles it took to perform the C statement was obtained from the difference between the time stamps and the average was calculated. The average was added to the CTAP annotated model by removing the estimate provided by CTAP

for that C statement (see line 8 of *Figure 7.18*). This procedure was followed for a total of 67 C statements.

The advantage of this annotated model is that the measurements can be performed for any image. As a result, is possible to make a comparison with different images and make a better evaluation of the CASSE Mixed-Level model.

7.5.6 Results

Table 7.1 presents the simulated time that the RTL specification and the three annotated models required to decode the *surfer.jpg* image. The first column shows the number of clock cycles taken with the RTL specification and the next three columns list the results for each of the annotated models.

The method used to measure the simulated time in both the RTL specification and the CASSE annotated models, is explained in detail at section 8.3.1.1.

RTL	**CASSE Mixed-Level model**		
	***mMIPS* Instrumentation**	**CTAP**	**CTAP with the compiler error correction**
6 086 649 cc	6 432 487 cc	2 036 313 cc	6 011 116 cc

Table 7.1: Accuracy of the different annotated models

As can be seen in *Table 7.1*, the correction approach provides the most accurate model (98.7%) since it estimates every C statement whereas the *mMIPS* instrumentation approach is lees accurate because it estimates larger parts of the C code. Even though the results of the *mMIPS* instrumentation are quite close to the simulated time of RTL, it is not as good as the correction option.

Ultimately, we decided to use the option which applies CTAP (with the correction for the compiler error) since it is provides a general annotated model which allows a complete evaluation with different images. The *mMIPS* instrumentation option does not allow this since it depends on the data in which the timing information has been obtained. In addition, it seems to be the most likely option in the context of a real design and also the most interesting one from a design/tool flow point of view.

7.6 Conclusions

The first step in achieving the implementation of the Mixed-Level model required an effort on modelling the application. To overcome that it was necessary to adapt the communication primitives to comply with the protocols of communication of CASSE. However, once developed they allow the communication between nodes of the model and can be used in any application.

The processor tiles of the model were easy and quick to develop since the tool provides generic architectural components which can be instantiated and interconnected in a plug and play fashion. In order to adapt the *mNoC* network to the model, it was necessary to develop an adaptor module which enables the transactions to be performed by changing the levels of abstraction.

The JPEG application was mapped easily and quick since the tool provides a direct support for application to architecture mapping.

For the timing annotation we explored several options, and finally chose to apply the CTAP tool since it provides a general annotated model allowing a complete evaluation and its accuracy is high. Since it was not possible to apply the correct compiler some manual annotations were required at specific parts of the application because the compiler used could not give the correct estimate. However, the problem was solved in a satisfactory manner.

Chapter 8

Comparison

8.1 Introduction

The previous chapters presented the *MiniNoC* platform and the CASSE Mixed-Level model. At this point we are able to perform the goal of this project, the comparison of the two platforms.

This chapter will first describe the different metrics identified for the comparison. Secondly, it will detail the measurements carried out in the two platforms in order to obtain the metrics and the comparison between them. The chapter ends with the conclusions reached based on the obtained results.

8.2 Metrics definition

This section presents the metrics identified for the comparison of the simulation of the two models, the difference between them and what applications are going to be used to obtain them.

The metrics defined for the comparison of the simulation of the *MiniNoC* (RTL specification) and the *CASSE* model are:

- Performance of the SystemC simulator for both the *CASSE* model and the *MiniNoC* platform. That is, the number of clock cycles that are simulated per second of computer time.

- Latency of processing each macro block used in the JPEG application. That is, the number of clock cycles between the start of processing a macro block and its completion, for each of the macro blocks.

- Latency of performing the *sc_send()* and *sc_receive()* functions for different message sizes. That is, the number of clock cycles it takes to perform the sc*_send()* and *sc_receive()* functions.

Whereas the performance of the simulator is chosen for the analysis of the global time accuracy and the simulation speed of the implemented model, the latency of each macro block will provide information about intermediate points of the simulation. This information allows an analysis to be made of the accuracy in several parts of the simulation and of when the data is processed. Therefore, these two metrics will provide an evaluation of speed of the implemented model, and the global and intermediate accuracy of the results.

The third metric allows validation of the architecture (P tiles, adaptors, NOC) and analysis of the *sc_send()* and *sc_receive()* communication primitives. Note that the metric includes not only computation latency but also communication time.

The JPEG decoder is the application selected to perform the first and the second metric. For the measurements of the last metric, a simple application which sends and receives message will be applied.

8.3 Metrics evaluation

This section presents the results obtained from the *MiniNoC* platform and the CASSE Mixed-Level model for each of the three metrics. Each section presents the way the metric was performed on each platform, the results and an analysis to evaluate the CASSE Mixed-Level model.

8.3.1 Performance of JPEG decoder

This metric measures the simulation performance of the JPEG decoder for the *MiniNoC* and the CASSE Mixed-Level timed model. The measurements presented are the number of clock cycles and the number of clock cycles per second that each platform took to execute the application. The first one will give us information of the global time accuracy of the Mixed-Level model, while the second provides the number of orders of magnitude higher the simulation speed of the CASSE model is than the RTL specification. In order to make a complete evaluation, the measurements have been taken for a total of seven images of different sizes, which gives more information about how the model behaves.

8.3.1.1 *MiniNoC* (RTL)

As explained in section 2.4, the *MIPS NOC* sources already provide the way to find out the last simulation time of the system. This information is provided by means of a time stamp function already added to the code, which is shown when the execution finishes through the standard output. So, it is no problem to obtain the simulated time required for decoding an image with the JPEG application.

On the other hand, the simulation time that the application needed is not provided directly by the platform. This information is necessary, since the aim is to find out the number of clock cycles per second. To achieve this he linux *time* command was used.

The *time* command runs the specified program with the given arguments. When the application finishes, *time* writes a message to standard output and gives timing

statistics about this program run. These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time and (iii) the system CPU time. In this case, the statistic most interesting is the one that gives the information about the user time because it provides the time that the application requiered to be performed in the host (see line 8 of *Figure 8.1*).

Once this command was included, the JPEG decoder was run on the *MiniNoC* platform as explained in section 2.4. When the application finished, the total real time used to complete the application was shown at the standard output and also the last simulation time of the system – supplied by the outcome of the linux *time* command (see line 11) and by the outcome of the SystemC *sc_time_stamp()* function (see line 8) .

```
1   //unnecessary lines omitted
2   …
3   dp_x1y0.netif data read 0 @ 58971630 ns
4   FINISHED x1y0 @ 58973610 ns
5   …
6   60860000/480000000 @ 60860010 ns PC: x0y0:0x28  x0y1:0x1ffc
    x1y0:0x28  x1y1:0x17375b4
7   FINISHED x0y1 @ 60868490 ns
8   ALL FINISHED @ 60868490 ns
9
10  real     48m56.034s
11  user     48m27.583s
12  sys      0m26.054s
```

Outcome of the *time* command

Figure 8.1: Output for the JPEG application for the surfer.jpg

The metric has been performed for different images. However, as an example, we have chosen the results of *surfer.jpg* image to shown the steps followed to obtain the metric.

With the simulated time and the period of the clock is 10 ns, it is possible to know the number of clock cycles to simulate the application. For example, the simulated time for the surfer was 60868490ns, therefore the number of clock cycles was:

$$\begin{matrix}\text{Number of clock cycles}\\ \text{to simulate the application}\end{matrix} = \frac{60868490}{10} = 6086849 \, clock\ cycles$$

The decoding of the *sufer.jpg* image on a 2x2 *mMIPS NOC* using the hardware simulator took **48m 27.583s** (2907.583s) on a Pentium IV 3.2GHz processor running GNU/Linux with 4096 MB of RAM (co5.ics.ele.tue.nl).
Finally, the number of clock cycles per second can be obtained by the division of the number of clock cycles and the simulation time spent on decoding the image:

$$\begin{matrix}\text{Number of clock cycles simulated}\\ \text{per second of computer time}\end{matrix} = \frac{6086849}{2907.583} = 2093{,}43 \, \frac{clock\ cycles}{sc} \cong 2{,}1 \, kHz$$

8.3.1.2 CASSE Mixed-Level model

Once the metric has been taken in the RTL specification, the next step is to perform it at CASSE model. The information required for the metric can be obtained directly from the standard output that CASSE provides when the simulation finishes. As an example, *Figure 8.2* shows the output of the decoding of the surfer image for the Mixed-Level model.

Observe that the data of the simulation performance shown at the end of the simulation has to be divided by the clock -10ns – to obtain the total number of clock cycles per second of computer time. The same process needs to be followed to obtain the total number of clock cycles. Finally, the simulation time data is directly provided by the tool.

```
"F:\casse1.0_release\examples\JPEG annotate_model\Release\gossip.exe"
 => Simulation started...
 => Starting task: step1
 => Starting task: step2
 => Starting task: step3
Processor Step2 up and running!
Processor Step3 up and running!
Processor Step1 up and running!

Done.
 => Finishing task: step1
 => Finishing task: step2
 => Finishing task: step3
SystemC: simulation stopped by user.


|------------------------------------------------ |
| 1 cycle = 1 ns
| Total cycles =          60111160 cycles
| Total time   =    112.9530 seconds
| Simulation performance = 532178.517 cycles/sc
|------------------------------------------------ |
 => Closing simulation...
 => Bye.
Press any key to continue
```

Figure 8.2: CASSE output for the annotated model after decoding the *surfer.jpg*

8.3.1.3 Results and Evaluation

The results obtained from both the *MiniNoC* platform and the CASSE model for the different images are presented in *Table 8.1*. To provide a more complete evaluation, the seven input images have different sizes. The first column lists the input images, which are ordered from largest to the smallest. The second column lists the different parameters of the metric such as the total number of clock cycles, simulation time and the number of clock cycles per second of computer time required for each model to decode the image. The next two fields represent an evaluation of the global accuracy and the speed-up of the Mixed-Level model compared to the RTL specification.

Input Image	Parameters	*MiniNoC* (RTL)	CASSE Mixed-Level
Gpyramid.jpg (80x60)	Clk cycles	22 097 001	20 878 720
	Simulation Time	319 m	390 sc
	Simulation Performance	1.1k	53.5k
	Accuracy	100%	94.4%
	RTL speedup (x times)	1	48.6
Tulip.jpg (38x48)	Clk cycles	14 137 610	13 302 198
	Simulation Time	116 m	249 sc
	Simulation Performance	2k	53.4k
	Accuracy	100%	94.1%
	RTL speedup (x times)	1	26.7
Pyramid.jpg (42x22)	Clk cycles	9 293 648	8 827 169
	Simulation Time	96 m	164 sc
	Simulation Performance	1.6k	53.5k
	Accuracy	100%	95%
	RTL speedup (x times)	1	33.4
Surfer2.jpg (30x32)	Clk cycles	7 184 050	6 592 221
	Simulation Time	77 m	129 sc
	Simulation Performance	1.5k	50k
	Accuracy	100%	91.7%
	RTL speedup (x times)	1	33.3
Surfer (32x24)	Clk cycles	6 086 649	6 011 116
	Simulation Time	48 m	113 sc
	Simulation Performance	2k	53.2k
	Accuracy	100%	98.7%
	RTL speedup (x times)	1	26.6

Pc.jpg (16x16)	Clk cycles	2 379 982	2 341 000
	Simulation Time	26 m	43 sc
	Simulation Performance	1.5k	54.1k
	Accuracy	100%	98.3%
	RTL speedup (x times)	1	36
Zw.jpg (8x8)	Clk cycles	902 963	894 544
	Simulation Time	9 m	17 sc
	Simulation Performance	1.6k	52.9k
	Accuracy	100%	99%
	RTL speedup (x times)	1	33

Table 8.1: Comparison for different size images

A closer look at the results of the largest image –*Gpyramid.jpg*, reveals that the accuracy is 94.4%. The simulation speed measured at RTL, is 1.1 Kcycles/sec, and at Mixed-Level, is 53.5 Kcycles/sec. Notice that even though the CASSE model supports a network which is described at RTL, the simulation speed went up to 48.6 Kcycles/sec. Observe also that it took 319 minutes in the RTL specification, while in the CASSE Mixed-Level it took only 6 minutes.

The table reveals that as the image becomes larger the number of clock cycles needed to decode it also increases. This fact is the reason why we included large images for the comparison since this requires more computation, and consequently the accuracy could change in a significant way. As shown in *Table 8.1,* this scenario does not occur in the CASSE Mixed-Level model, since the accuracy is near to 95% and almost constant for the larger images.

This trend is shown at *Figure 8.3*, which illustrates the total number of clock cycles required for each of the seven images for both models. Note that the error remains almost constant for all the different images in spite of the fact that larger images require more computation.

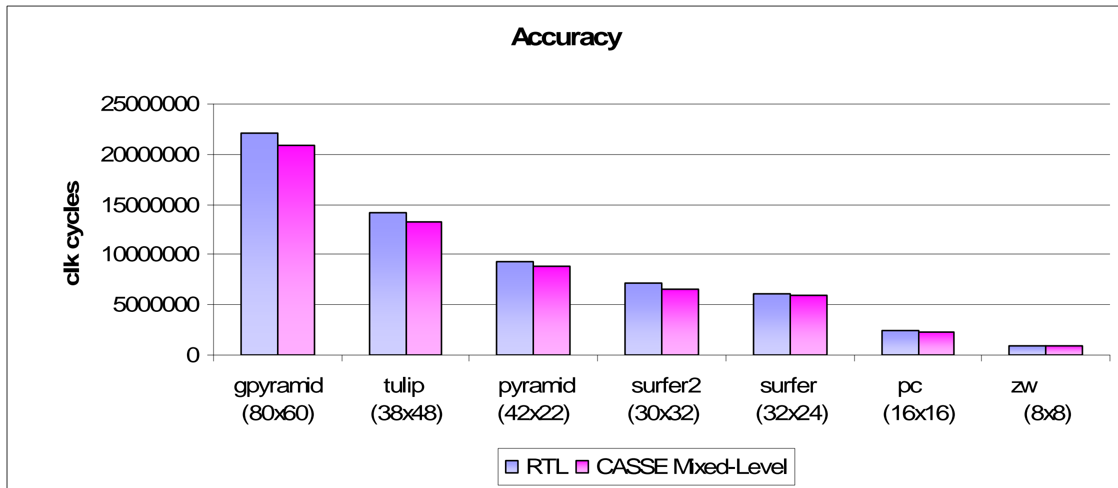


Figure 8.3: Accuracy evaluation

Figure 8.4 presents the simulation time in seconds needed to execute JPEG for the decoder of the seven images. As depicted in the logarithmic graphic, the Mixed-Level model simulation time is at least one order of magnitude faster than the RTL simulation.

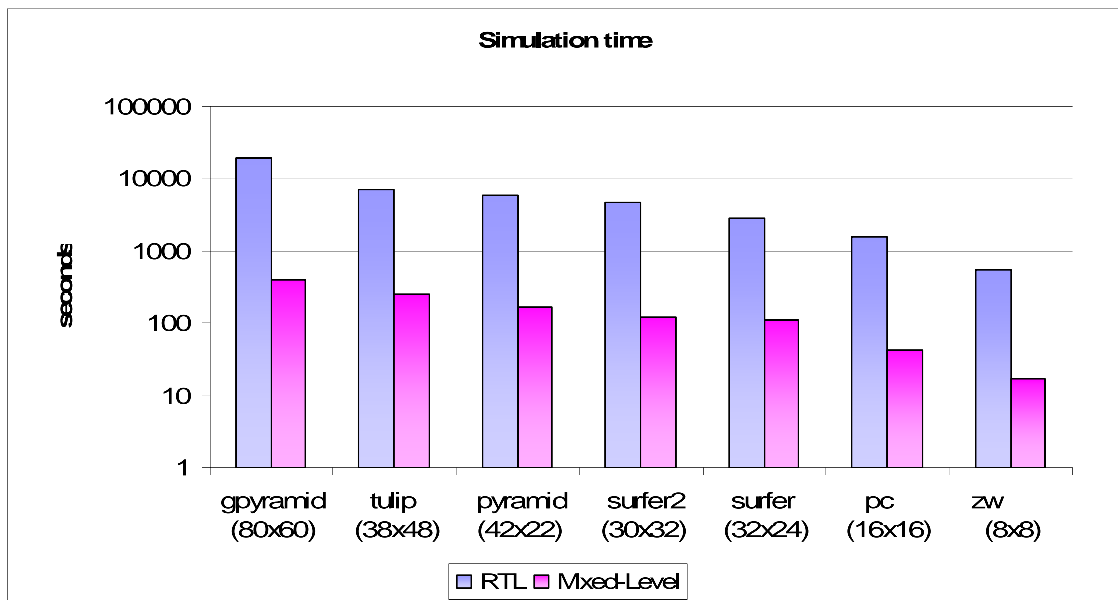


Figure 8.4: Simulation time comparison

The experimental results reveal that the simulation speed is increased by a factor of 34 on average compared to the RTL execution. *Figure 8.5* depicts on a logarithmic scale the simulation performance for the different images. Observe that 1.1kcc/sec is the lowest simulation speed for the RTL specification, which corresponds to the largest image.

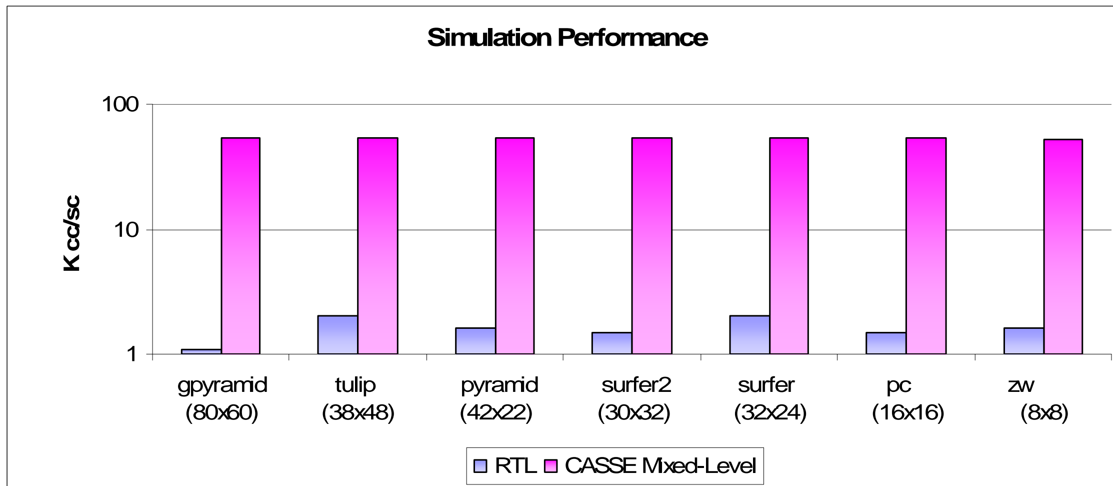


Figure 8.5: Simulation speed comparison

To sum up, from the experimental results, the implemented model is 34 times faster than RTL model while maintaining 96% of accuracy on average.

8.3.2 Latency of processing each macro block

As explained at section 5.2, the JPEG process has been divided up into three nodes as represented in *Figure 5.1*. The input image applied to the decoder is separated into in a number of blocks which are grouped in macro blocks that describe a certain region of the image. The splitting process is performed at node 1, and then the macro blocks are sent as blocks through the *mNoC*. The blocks are received and processed at node 2 and then forwarded to node 3, where they are processed and reordered into a macro block.

The latency tries to find out the number of clock cycles between the start of processing a macro block and its completion, for the total of the macro blocks. That is, the number of the clock cycles that are required from when the node 1 processes the macro block until the macro block is received and processed at node 3. Since the packets require to go through node 2 until they reach to node 3, the metric involves both communication and computation time.

In addition, a comparison will also be performed with the timing information about when each macro block is processed. Whereas the latency provides information for performing an analysis of the accuracy at intermediate moments of the application, with the timing information, it is possible to evaluate when the data is processed. These two parameters will provide a complete analysis of the intermediate performance features of the model.

As for every metric, we will first describe how the measurements have been taken in the two models, and then the results and evaluation will be presented. In order to make a complete evaluation, the measurements have been made for a total of three images of different sizes.

8.3.2.1 *MiniNoC* (RTL)

To obtain the latency, the *TIME_STAMP* instruction has been used to find out the measurements. As explained in section 6.2, this instruction was performed to show the current time of the system.

The first step was to add two time stamps one at the node 1 and the other node 3, where the application starts and finishes processing the macro block. Once the time stamps were added, the JPEG decoder was run on the *MiniNoC* platform as explained in section 2.4. As any other application, the decoder was compiled with the LCC C compiler and then run in the SystemC simulator for the *mMIPS NOC* (see *Figure 2.5*). When the application was running the timing information was shown in the standard output and the latency was obtained from the difference between the two time stamps. This process has been followed for each macro block. An example of how the latency was measured for the *surfer.jpg* image is shown in the following figure:

```
1   //unnecessary lines omitted
2   …
3   network2x2.x0y1.yrouter.dqueue sent flit 0x20003 @2975110 ns
4   dp_x0y1.netif received data 3 @ 2975120 ns
5   dp_x0y1.netif data read 3 @ 2975390 ns
6   dp_x0y1.netif data read 3 @ 2975620 ns

7   dp_x0y0.mips.time stamp.......................... 2979980 ns

8   dp_x1y0.netif data read 1 @ 9894010 ns

9   dp_x0y1.mips.time stamp.......................... 9894380 ns
    …
```


Figure 8.6: Extract of the standard output

8.3.2.2 CASSE Mixed-Level model

For the CASSE Mixed-Level model the latency was also obtained by adding time stamps at the node 1 and node 3, by means of the SystemC *sc_time_stamp()* function. When the application ran the timing information was shown in the CASSE standard output and the latency was obtained from the difference between the two time stamps.

8.3.2.3 Results of the latency of processing each macro block

The experiments were performed for a total of three images. Since their sizes are quite different, the number of macro blocks required to be processed also varied between them. This allows a complete comparison with a number of macro blocks which differ from 4 to 12, and for which the starting and ending points occur in a wide range of instants. The results give relevant information about how the model behaves with different images.

As explain above, it is possible to obtain the time the decoder took to process a macro block, by the subtraction of the time when the macro block started to be processed from the time when is completely processed. Since the objective is the number of clock cycles, it is possible to obtain this by dividing the previous result by 10 since the clock is 10ns. This process has been followed for each macro block for three images in both models.

8.3.2.3.1 Surfer image

The results obtained for the surfer image in the RTL specification and the Mixed-Level model, are indicated in *Tables 8.2* and *8.3.* The first column indicates the number of each macro block of the surfer image. The second column lists the time stamp when the macro block started to be processed in node 1. The third column lists the moment that the macro block is completely processed at node 3. The last column contains the total number of clock cycles needed to process each macro block.

Macro block	Start processing	End processing	Clock cycles
0	2 979 980 ns	9 894 380 ns	**691 440**
1	6 815 020 ns	14 512 450 ns	**769 743**
2	11 434 210 ns	19 130 920 ns	**769 671**
3	16 051 930 ns	23 750 820 ns	**769 892**
4	20 671 220 ns	28 370 820 ns	**769 960**
5	25 291 240 ns	33 146 070 ns	**785 483**
6	30 072 350 ns	37 765 290 ns	**769 294**
7	34 686 000 ns	42 385 690 ns	**769 969**
8	39 305 970 ns	47 006 020 ns	**770 232**
9	43 925 450 ns	51 625 630 ns	**770 018**
10	48 545 750 ns	56 245 400 ns	**769 965**
11	53 165 000 ns	60 866 140 ns	**770 114**

Table 8.2: Latency of processing each macro block for the RTL for the *surfer* image

The results lead to the conclusion that each macro block is processed in approximately 700k clock cycles. Note that the latency of the first macro block is lower because the pipeline of the network is empty, so the latency of the rest of macro blocks also includes the time waiting at the buffers for the previous macro block to finish its processing.

Macro block	Start processing	End processing	Clock cycles
0	2 990 930 ns	9 784 250 ns	**679 332**
1	6 953 840 ns	14 305 230 ns	**735 139**
2	11 474 820 ns	18 826 160 ns	**735 134**
3	15 995 740 ns	23 347 190 ns	**735 145**
4	20 516 760 ns	27 868 190 ns	**735 143**
5	25 037 650 ns	32 713 770 ns	**767 612**
6	29 883 320 ns	37 318 400 ns	**743 508**
7	34 487 930 ns	41 839 230 ns	**735 130**
8	39 008 840 ns	46 360 330 ns	**735 149**
9	43 529 780 ns	51 069 210 ns	**753 943**
10	48 238 740 ns	55 590 140 ns	**735 140**
11	52 759 740 ns	60 111 070 ns	**735 133**

Table 8.3: Latency of processing each macro block for the CASSE Mixed-Level model for the *surfer*

To allow an analysis of *Tables 8.2, 8.3*, *Figure 8.7* shows the starting and ending of each macro block. Note that the implemented model is quite accurate since it starts and ends processing at the same order of magnitude.

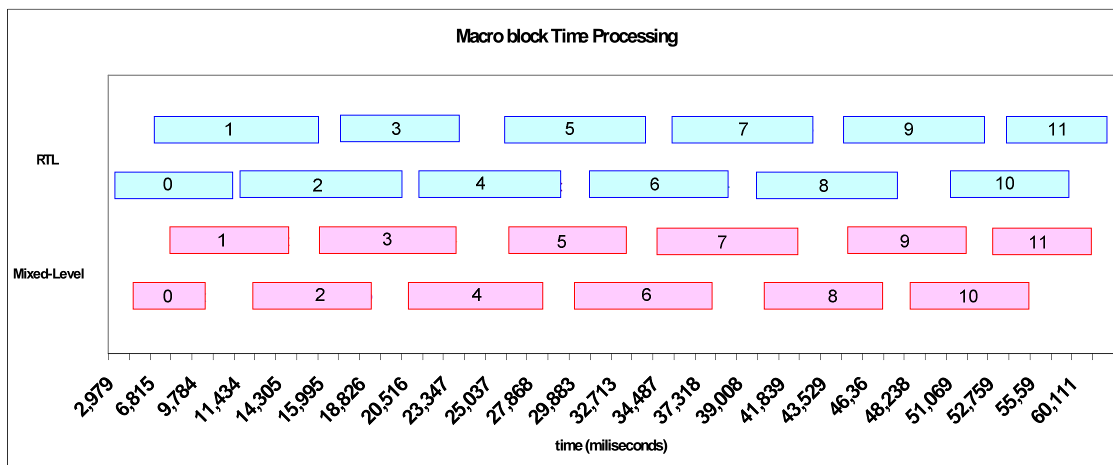


Figure 8.7: Comparison of the macro block time processing for the *surfer* image

Observe that the approach at the starting point of the processing the first macro block is very close to the RTL (99,6% accurate). This fact is important since a lot of computation process is required at node 1 before any first macro block is processed. Moreover, the computation at node 1 has high data dependence, i.e. depending on the input image it requires more or less computation time. Therefore, the error introduced in this point is useful since it estimates the loss of accuracy of the model. Observe that after correcting that error, each macro block of the image would start at almost the same moment as it does on the RTL specification.

With the latency information of the previous tables, an evaluation of the accuracy has been made and shown in *Table 8.4*. The first column lists the number of macro blocks of the image. The second and third columns represent the number of clock cycles that both models required to process each macro block. The next column shows the difference between the number of clock cycles between the two models. Finally, the last column shows the accuracy of the Mixed-Level model for each macro block.

Macro block	RTL	Mixed-Level	Clk Cycle Diff	Accuracy (%)
0	691 440	679 332	12 108	98,4
1	769 743	735 139	34 604	95,5
2	769 671	735 134	34 537	95,5
3	769 892	735 145	34 747	95,4
4	769 960	735 143	34 817	95,4
5	785 483	767 612	17 871	97,7
6	769 294	743 508	25 786	96,4
7	769 969	735 130	34 839	95,4
8	770 005	735 149	34 856	95,4
9	770 018	753 943	16 075	97,9
10	769 965	735 140	34 825	95,4
11	770 114	735 133	34 981	95,4

Table 8.4: Comparison of the macro block's latency for the *surfer* image

A closer look to the *Table 8.4* reveals that the number of clock cycles varies between from 12k and 34k. As a result, the implemented model can achieve an average accuracy of 96,1%.

Figure 8.8 represents the comparison of the latency of each macro block for the two models. Note that the approach of the Mixed-Level is quite close to the RTL model, by following the same trend for the different measurements. Moreover, it also includes the pipeline effect of the network that occurs at the first macro block of the RTL model.

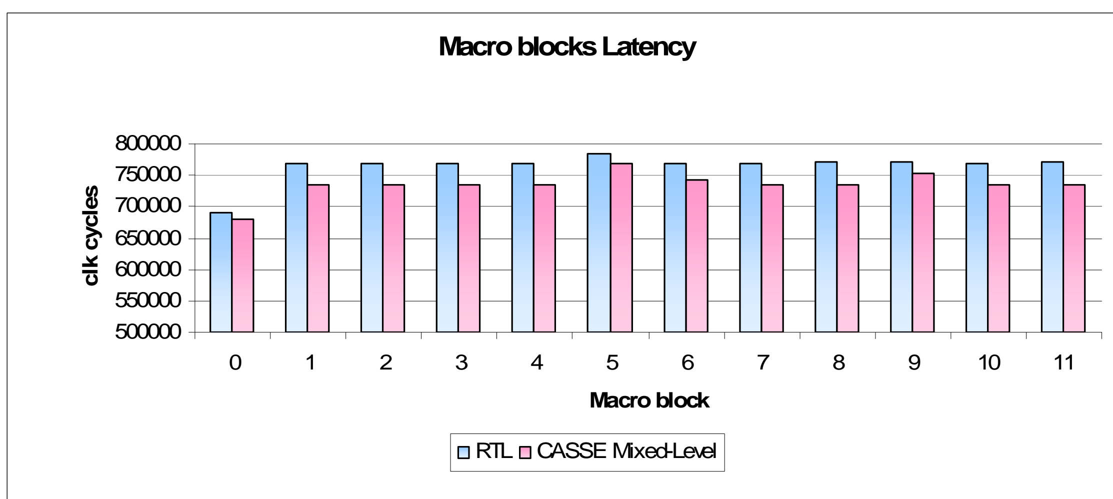


Figure 8. 8: Comparison of the latency of the macro blocks for the *surfer* image

8.3.2.3.2 Pyramid image

The experiments performed with the surfer image have been repeated for a 32x24 pixels colour image, the pyramid. The image is split in 6 macro blocks and the latency results for the two platforms are presented in *Tables 8.5* and *8.6*.

Macro block	Start processing	End processing	Clock cycles
0	11 075 210 ns	27 025 370 ns	**1 595 016**
1	18 068 080 ns	41 561 960 ns	**2 349 388**
2	32 605 360 ns	55 101 660 ns	**2 249 630**
3	47 141 220 ns	67 817 570 ns	**2 067 635**
4	60 681 920 ns	80 533 680 ns	**1 985 176**
5	73 398 940 ns	92 933 300 ns	**1 953 436**

Table 8.5: Latency of processing each macro block for the RTL model for the *pyamid* image

Macro block	Start processing	End processing	Clock cycles
0	8 329 650 ns	23 448 730 ns	**1 511 908**
1	15 610 380 ns	37 131 540 ns	**2 152 116**
2	29 293 180 ns	50 252 830 ns	**2 095 965**
3	42 976 040 ns	62 995 990 ns	**2 001 995**
4	56 087 280 ns	75 739 030 ns	**1 964 175**
5	68 840 450 ns	88 271 600 ns	**1 943 115**

Table 8.6: Latency of processing each macro block for the Mixed-Level model for the *pyamid* image

Figure 8.9 shows the start and finish of each macro block of the pyramid image. Observe that the approach at the starting point of the processing the first macro block is the one which differs most from the RTL, even though the accuracy is 75,2%.

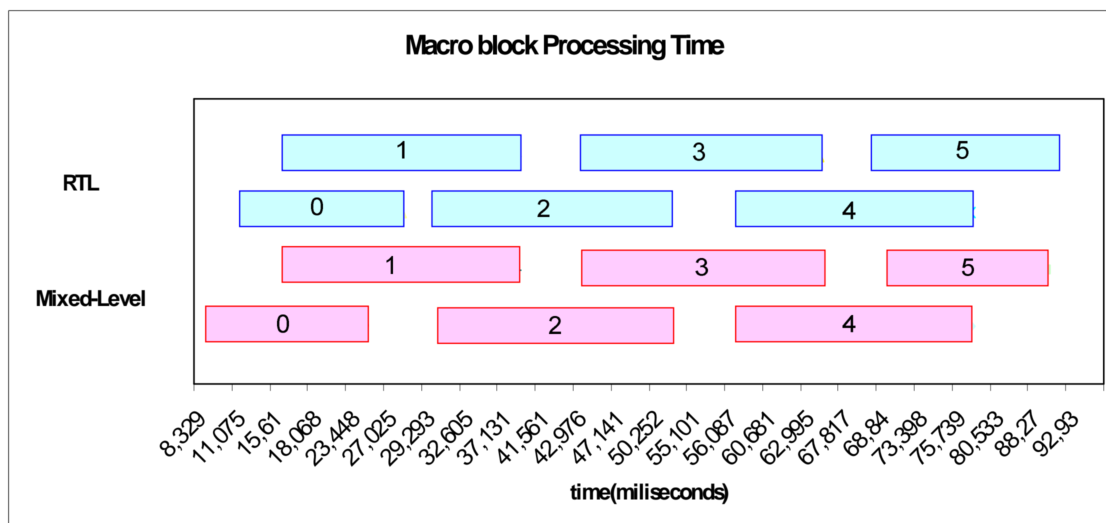


Figure 8.9: Comparison of the macro block time processing for the *pyramid* image

Table 8.7 lists the comparison between the two latencies. The third column which indicates the results of the *MiniNoC* platform reveals that the number of clock cycles required for processing each macro block is on average 2 million. By comparing with the results of the Mixed-Level model, is easy to estimate that they are rather closer: 1.9 million.

Image	Macro block	RTL	Mixed-Level	Clk Cycle Diff	Accuracy (%)
Pyramid.jpg (42x22)	0	1 595 016	1 511 908	83 108	94,7
	1	2 349 388	2 152 116	197 272	91,6
	2	2 249 630	2 095 965	153 665	93,1
	3	2 067 635	2 001 995	65 640	96,8
	4	1 985 176	1 964 175	21 001	98,9
	5	1 953 436	1 943 115	10 321	99,4

Table 8.7: Comparison of the macro block's latency for the *pyramid* image

This fact is clearly shown in *Figure 8.10* which represents the two models. To be precise the number of clock cycles differs between 0k and 200k, which indicates that the average accuracy of implemented model for the simulations with the pyramid image is around 95,7%.

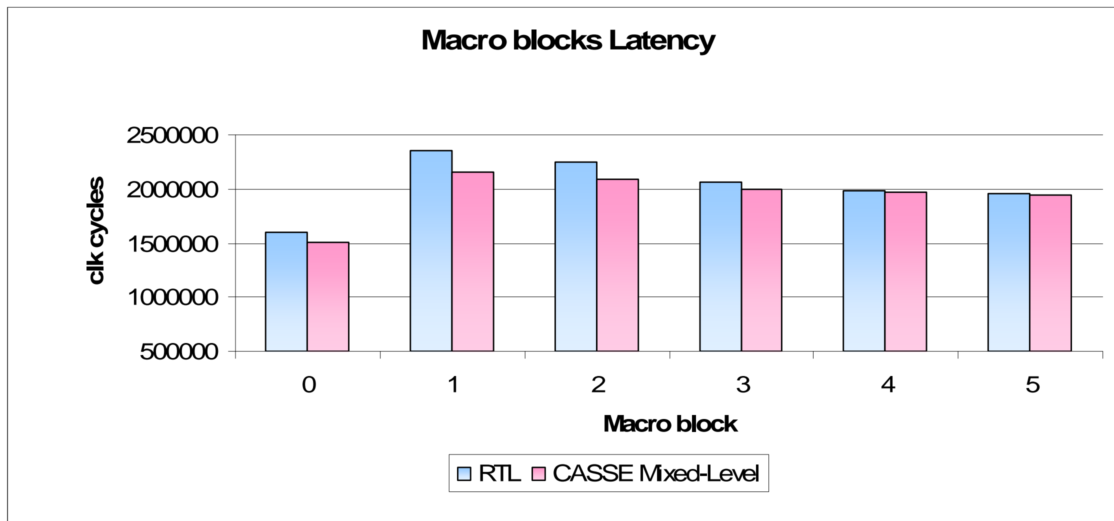


Figure 8.10: Comparison of the latency of the macro blocks for the *pyramid* image

8.3.2.3.3 Surfer2 image

The surfer2 image is the surfer image doubled in size, 30x32, and split into 4 macro blocks. The results obtained on the RTL specification and the Mixed-Level model, are indicated in the *Tables 8.8* and *8.9*.

Macro block	Start processing	End processing	Clock cycles
0	11 069 460 ns	28 408 980 ns	**1 733 952**
1	19 456 940 ns	43 244 880 ns	**2 378 794**
2	34 287 530 ns	57 616 210 ns	**2 332 868**
3	49 038 090 ns	71 839 620 ns	**2 280 153**

Table 8.8: Latency of processing each macro block for the RTL for the *surfer2* image

Macro block	Node 1	Node 3	Clock cycles
0	8 329 600 ns	24 722 080 ns	**1 639 248**
1	16 883 730 ns	38 685 540 ns	**2 180 181**
2	30 847 210 ns	53 362 690 ns	**2 251 548**
3	44 712 110 ns	65 922 120 ns	**2 121 001**

Table 8.9: Latency of processing each macro block for the CASSE Mixed-Level model for the *surfer2*

As for the other images, the processing time of the image has been represented in *Figure 8.11*. The accuracy of the start of the first macro is 75,2%, which is the same as for the pyramid image.

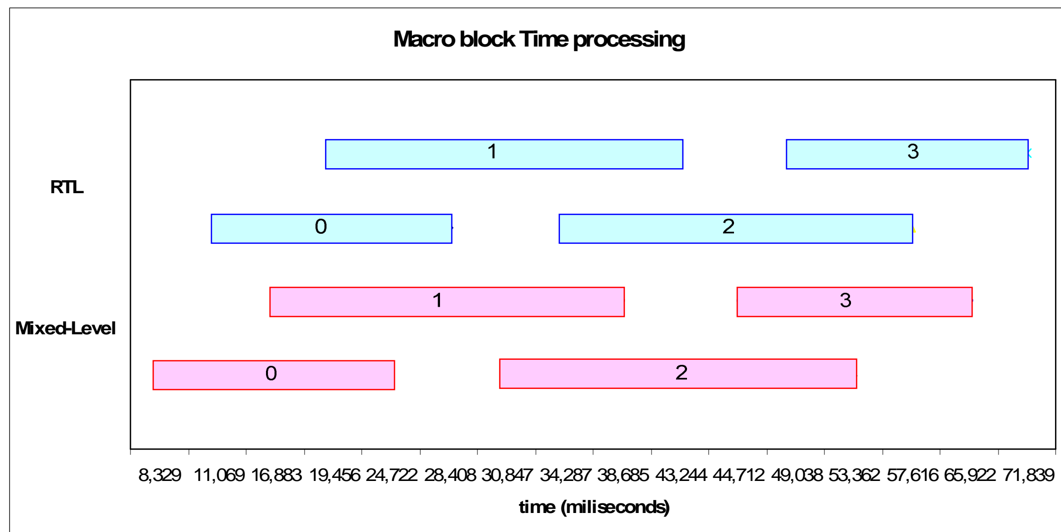


Figure 8.11: Comparison of the macro block time processing for the *surfer2* image

Table 8.10 represents the comparison of the latency results for the two models. This table indicates that the latency accuracy achieved by the implemented model is approximately the 93.9%.

Image	Macro block	RTL	Mixed-Level	Cycle Diff	Accuracy (%)
Surfer2.jpg (30x32)	0	1 733 952	1 639 248	94 704	94,5
	1	2 378 794	2 180 181	198 613	91,6
	2	2 332 868	2 251 548	81 320	96,5
	3	2 280 153	2 121 001	159 152	93

Table 8.10: Comparison of the macro block's latency for the *surfer2* image

As for the other images, the latency of the macro blocks has been represented in a chart (*Figure 8.12*) where the difference between both results can be observed, and that both tendencies are quite similar.

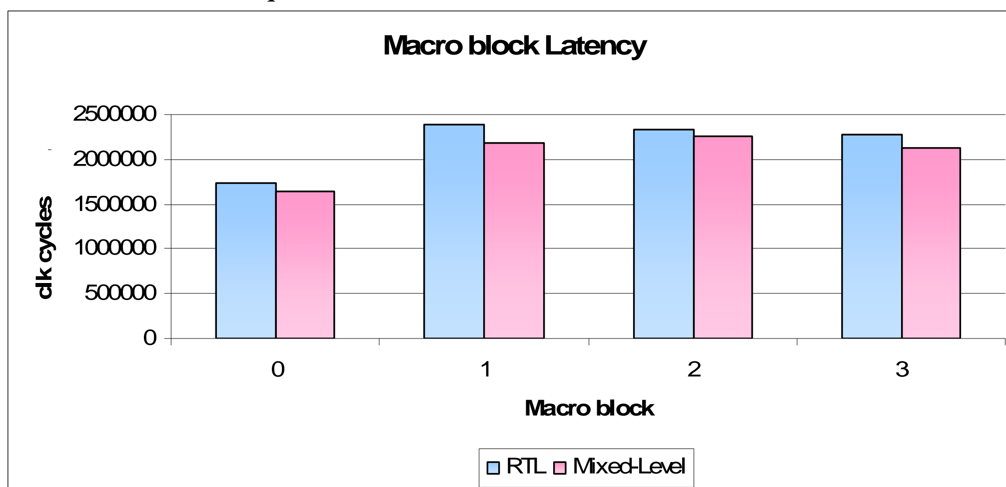


Figure 8.12: Comparison of the latency of the macro blocks for the *surfer2* image

8.3.3 Total latency of sending one specific message

The aim of this metric is to find the total number of clock cycles between completing the reception of the message and the start of sending the message.

8.3.3.1 *MiniNoC* (RTL)

The first step was to code a simple application and to run it in the *MiniNoC*. The application- name *Test* - sends a message from the x0y0 node to the x1y0 node. Thus, only a *sc_send()* function is executed in the (0,0) node, and only a *sc_receive()* is executed in the (1,0) node (see lines 9 and 13 of *Figure 8.14*). The message is split into word packets of 4 bytes that are sent to the network interface and then forwarded through the network. The *sc_receive()* function collects all the words and rebuilds the message by putting them all together. Since the only words in the network come from (0, 0) node, no contention occurs (see *Figure 8.13*).

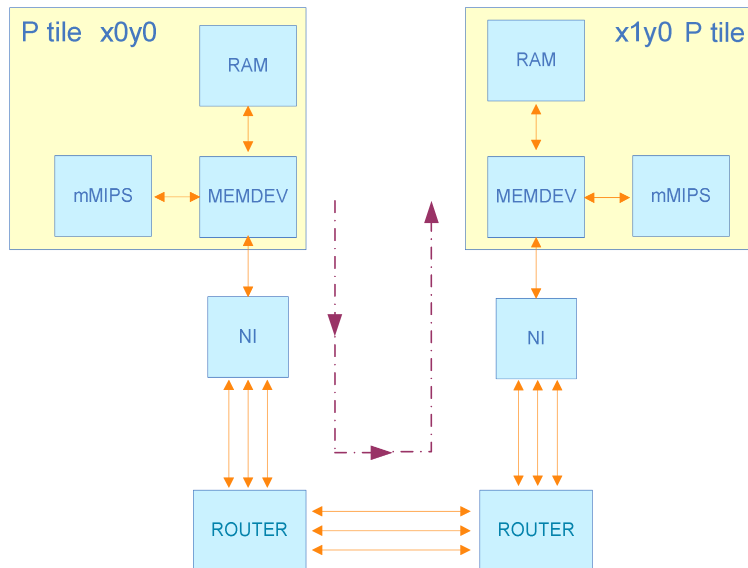
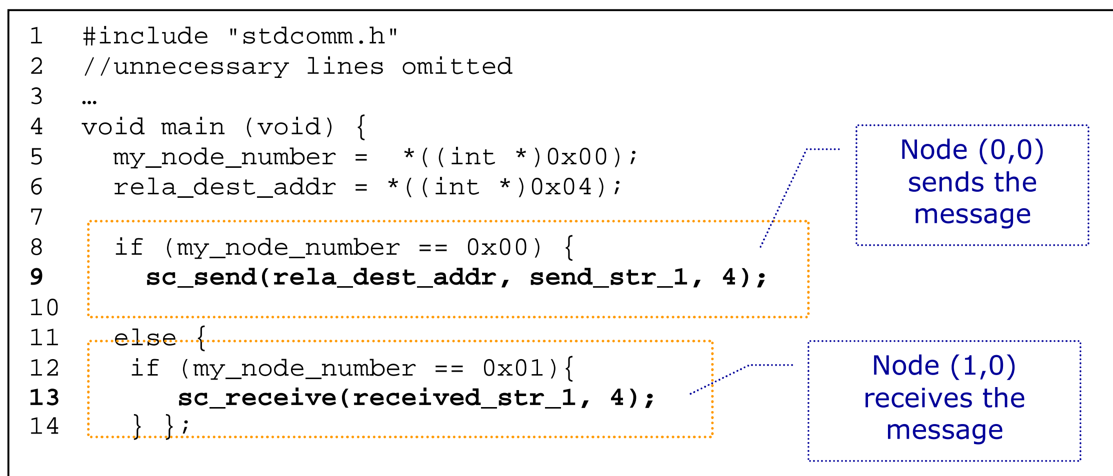


Figure 8.13: Nodes used in the application

In this metric the *TIME_STAMP* instruction was applied to find out the total number clock cycles that each *sc_send()* and *sc_receive()* function required. The measure was performed by adding the instruction before and after the function. Once the simulation was finished the latency was obtained from the difference between the two time stamps. The use of the instruction allows adding accuracy to the measure, since the interest is only in the latency of the functions (not the application).



```
1   #include "stdcomm.h"
2   //unnecessary lines omitted
3   …
4   void main (void) {
5     my_node_number =  *((int *)0x00);
6     rela_dest_addr = *((int *)0x04);
7
8     if (my_node_number == 0x00) {
9       sc_send(rela_dest_addr, send_str_1, 4);
10
11    else {
12     if (my_node_number == 0x01){
13       sc_receive(received_str_1, 4);
14     } };
```


Figure 8.14: *Test* application performed for 1 packet message

8.3.3.2 CASSE Mixed-Level model

As for the *MiniNoC,* a simple application was performed, in which a message is sent from the x0y0 node to the x1y0 node. Therefore, only a *sc_send()* function is executed at (0,0) node, and only a *sc_receive()* is executed at (1,0) node. The words in the network come from x0y0 node, therefore no contention occurs (see *Figure 8.15*).

The measure was performed by adding time stamps at the end of the *sc_send()* and *sc_receive()* functions of the application. Once the simulation was finished the latency was obtained at the standard output.

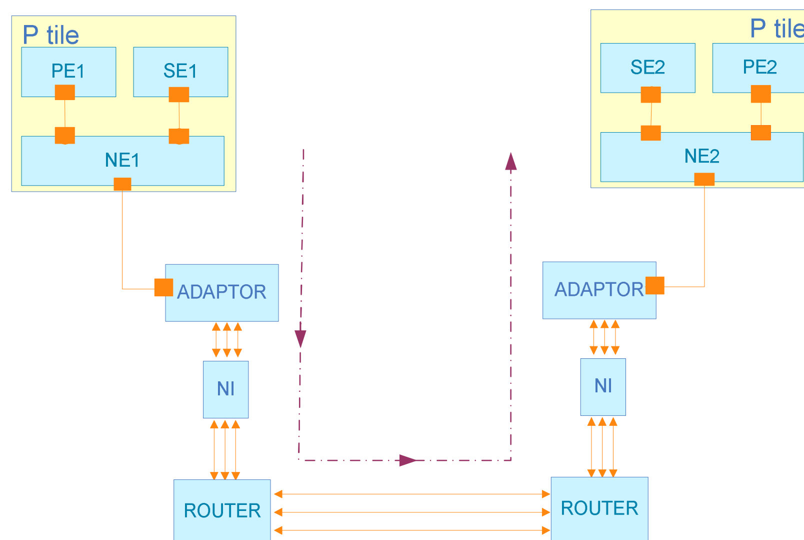


Figure 8.15: P Tiles used in the application

8.3.3.3 Results of the latency of sending one specific message

The measurements have been taken for messages of 1 word (4 bytes), 2 (8 bytes), 4 (16 bytes), 8 (32 bytes), 16 (64 bytes), 32 (128 bytes), and 64 (256 bytes). *Table 8.11* indicates the latency of the *sc_send()* primitive measured for the *MiniNoC* platform and the CASSE Mixed-Level model.

Primitive	Models	1	2	4	8	16	32	64
sc_send	RTL	311	571	1091	2131	4211	8371	16691
	Mixed-Level	317	597	1157	2277	4517	8997	17957

Table 8.11: Latency of the *sc_send()* primitive for different size message at the both platforms

The first row represents the number of words sent in different size messages. The second and last row list the number of clock cycles that the *sc_send()* function required to perform each message in both platforms.

By having a look to *Table 8.11*, it seems that the error introduced increases as the number of packets sent in a message rises. By making an analysis of the relative error introduced in each of the different sizes messages, the conclusion reached is that in a global perspective the error slightly increases. See *Table 8.12*. This is because the code size executed increases does with the number of packets, as a result the error also increases.

Packets	1	2	4	8	16	32	64
Relative Error	$1.9 \cdot 10^{-2}$	$6 \cdot 10^{-2}$	$6.8 \cdot 10^{-2}$	$7.2 \cdot 10^{-2}$	$4.5 \cdot 10^{-2}$	$7.4 \cdot 10^{-2}$	$7.5 \cdot 10^{-2}$

Table 8.12: Relative error introduced in the *sc_send()* primitive latency of the Mixed-Level model

Note that at the RTL implementation, as the number of packets in a message doubles, the latency also doubles in clock cycles. This fact also occurs at the implemented Mixed-Level model. Therefore, *Table 8.11* shows that the tendencies of the two latencies are similar. This feature can be observed in *Figure 8.16,* where the latency for the *send* primitives measured in the two platforms is represented on a logarithmic scale.

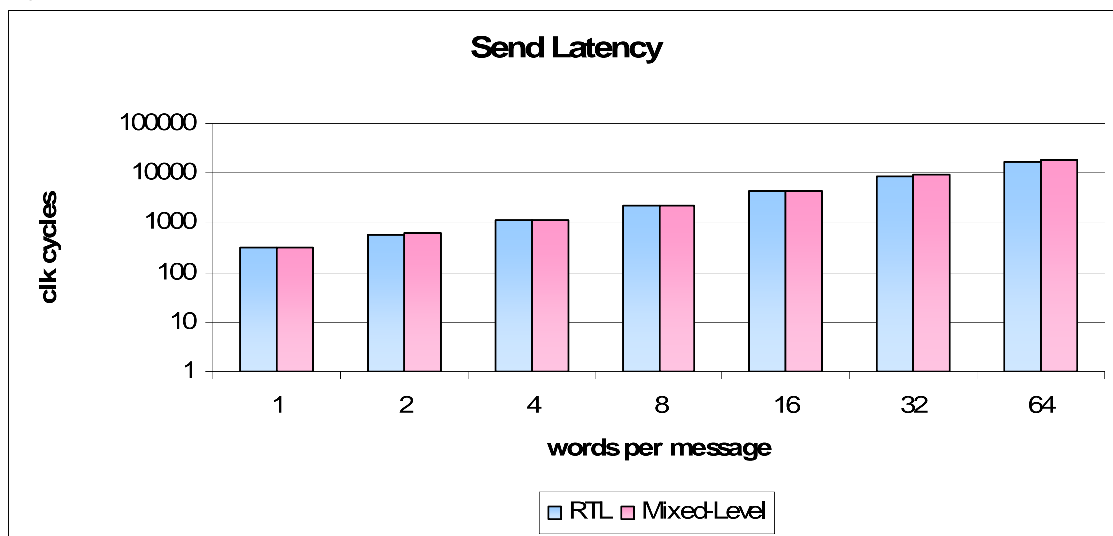


Figure 8.16: Comparison of the latencies of the *sc_ send()* primitive in both models

The same analysis has been performed at the *sc_receive()* primitive, the results of which are shown in *Table 8.13* for different message sizes.

Primitive	Models	1	2	4	8	16	32	64
sc_receive	RTL	490	778	1260	2318	4387	8572	16895
	Mixed-Level	495	771	1334	2449	4690	9172	18136

Table 8.13: Latency of *sc_receive()* for different size message at the both platforms

Note that the *sc_receive()* function requires more than the *sc_send()* function since it requires all the packets to be put together and the message rebuild, so some extra time is spent in that. This feature is also included at the latency of the implemented model. However, as for the *sc_send()* primitive the number of clock cycles required to receive a message doubles, as the size of the message also doubles. This relationship can be observed in *Figure 8.17*. The chart also shows the small difference between the two latencies, represented on a logarithmic scale.

By performing the same analysis of the relative error to the *sc_receive()* primitive, we reached to the same conclusion: the error increases slightly with the size of the message.

Packets	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Relative Error	$1 \cdot 10^{-2}$	$0.8 \cdot 10^{-2}$	$5.8 \cdot 10^{-2}$	$5.6 \cdot 10^{-2}$	$6.9 \cdot 10^{-2}$	$6.9 \cdot 10^{-2}$	$7.3 \cdot 10^{-2}$

Table 8.14: Relative error introduced in the *sc_receive()* primitive latency of the Mixed-Level model

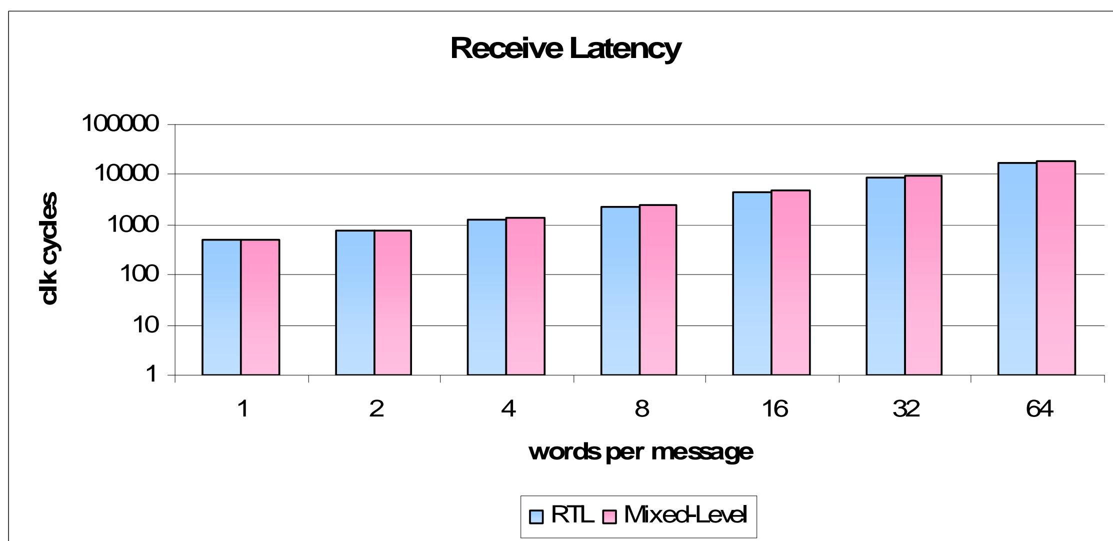


Figure 8.17: Comparison of the latencies of the *sc_ receive()* primitive in both models

8.4 Conclusions

From the experimental results, the CASSE Mixed-level model is on average 34 times faster than the RTL specification while maintaining 96% accuracy.

The analysis of intermediate features reveals that the accuracy at different parts of the application is on average 95.2%. Furthermore, the time at which the data is processed on the implemented model almost exactly coincides with the time on the RTL specification. The last conclusion from this analysis is that the loss of global accuracy is introduced at node 1 due to its high data dependence. Therefore, depending on the input image it requires more or less computation time in this way an error is introduced into the model.

Finally, we can conclude that the suitability of the model has been validated since it faithfully represents the system feature. The functionality of communication primitives presents the same as the original *MiniNoC* platform with an inaccuracy lower than 7%.

Chapter 9

Conclusion and future work

This chapter contains conclusions formulated based on the obtained results, recommendations and proposals for future work to continue with the work of this Master Thesis.

9.1 Conclusions

The main objective proposed for this Master Thesis, so and as it is described in section 1.2, it consisted of the development of a mixed model between two levels of abstraction RTL and TLM that emulated the *MiniNoC* platform applying the CASSE environment. This mixed model required to be compared itself with the reference implemented in SystemC RTL in terms of speed of simulation, numbers of cycles of execution and modeling effort. As is described through this project the main objective has been achieved.

Modeling of the *MiniNoC* platform was very easy since the CASSE environment allows describing the system as a modular composition of predefined elements that can be created and configured by a textual description file. This text file allows performing a simple description of the elements that compose the architecture and its configuration, with no need to write manually no line of code in SystemC.

The greater effort modeling the platform was due to the creation of adapters that allowed the communication between the processors nodes and the *NoC* network, since the nodes processors (TLM) and the *NoC* network (RTL) were described in different levels of abstraction. Nevertheless, most of the work of this Master Thesis was not spent at the architecture modeling but in the annotation to the JPEG tasks of the duration information of the execution. The problems arose with the *mMIPS* LCC compiler that did not make possible to use the CTAP tool. Even so, these problems were overcome in a satisfactory manner by using the compiler MIPS GCC processor instead and by replacing the software instructions from the JPEG code and the *stdcomm* library.

According to the results obtained from the comparison, the CASSE Mixed-Level model in combination with tool CTAP produced results of 96% of average accuracy in the execution time and an improvement in the simulation speed on an average factor of 34. That is, with almost the same number of cycles of execution the simulations in CASSE are 34 times faster, even considering that part of the platform is still described at RTL level which slows down the simulation. In accordance with the obtained results it is possible to be concluded that tool CTAP is a perfect combination with the code emulation technique used in CASSE, and that allows removing the instruction set simulators (ISS) without a big repercussion at the simulation.

The most important conclusion of this project is that the combination of the CASSE and CTAP tools allows easy and fast design space exploration of *NoC-based MPSoC*. This fact is because of the fast simulations that takes place maintaining a good level of precision in the results and modeling efficiency. In addition, this project contributes to demonstrate that the time dedicated to the functional validation of complex platforms can be reduced using dramatically technical of abstraction via models SystemC TLM. This improvement is present even when for multi-level models are created. This enables the engineers to verify the implementation of specific blocks of hardware at level RTL, whereas the rest of the system remains at a level of greater abstraction by means of models described in TLM.

9.2 Recommendations

For possible future research continuing the present work, the following suggestions are made:

- Provide the *mMIPS* LCC C compiler to support the standard STABS debugging format. The modification would allow the compiler to be applied in CTAP.

- Extend the instruction set of the *mMIPS* processor with the implementation in hardware of the operations that currently need to be done by software. The extension would optimize the simulations of the RTL specification.

9.3 Future work

The future work proposed is to substitute the NOC component of the Mixed-Level model with a new NOC component modelled at a high level of abstraction (TLM). In this way, the new CASSE model will be completely described at TLM, expecting a huge simulation speed-up.

The implementation proposed for the new NOC component is to be composed of two basic modules: a network interface and a router (*Figure 9.1*). Once the two modules are developed, the idea is to connect them as the *mNoC* network of the *MiniNoC* platform (*Figure 2.4*).

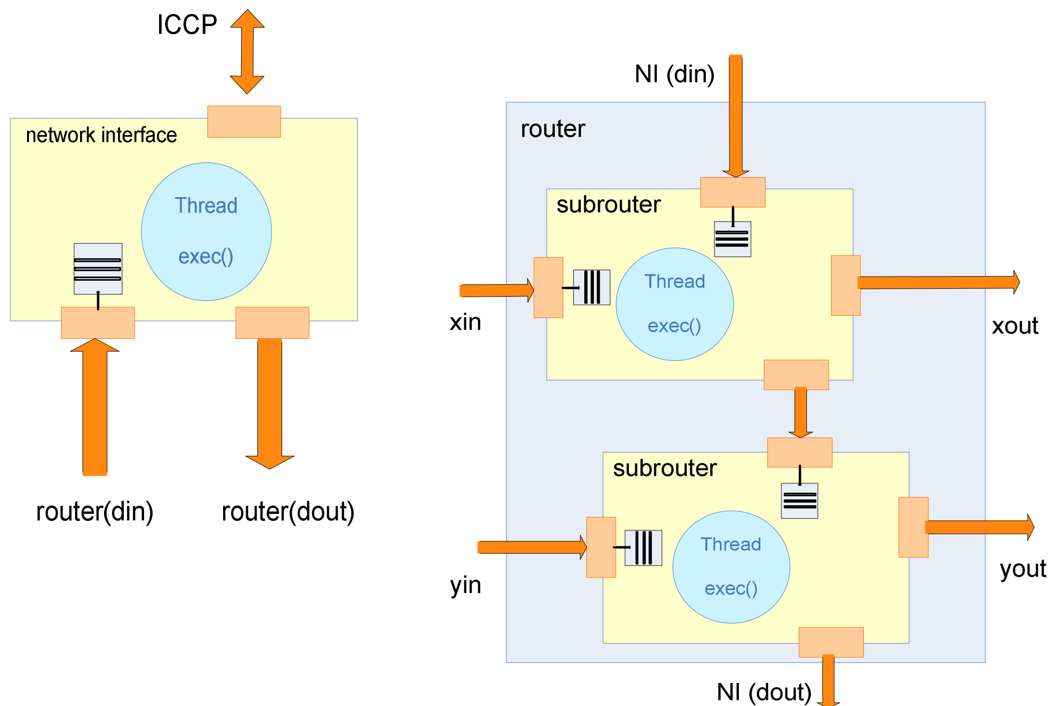


Figure 9.1: Modules proposed for future work

The modeling of the network interface module needs to develop a thread to control the transactions on both ways (processor tiles and routers) and also needs to add a buffer. The buffer will be placed at the input ports to manage the input data.

The router will be modelled as the original implementation of the *MiniNoC* routers. It is composed with two subrouters with have the same functioning. Each one contains a thread that controls the data by managing it by means of buffers. The priority is established by a round robin algorithm as in the original *mNoC* the router.

An important feature is that when data does not need to be managed, all the threads will be suspended during that the time, which allows the speeding-up the simulation since no clock cycles are consumed.

Bibliography

[1] R. Bergamaschi, W. R. Lee, “Designing Systems-on-Chip Using Cores“, In Proc of DAC, 2000.

[2] W. Cesário et al, “Component-Based Design Approach for Multicore SoCs,” In Proc. of DAC, 2002.

[3] D. Burger, J. R. Goodman, “Billion-Transistor Architectures: There and Back Again”, In IEEE Computer, March 2004.

[4] S. Naffziger, T. Grutkowski, B. Stackhouse, “The Implementation of a 2-core Multi-Threaded Itanium® Family Processor”, In Proc of ISSCC, 2005

[5] J. Henkel, W. Wolf, S. Chakradhar. “On-chip networks: A scalable, communication-centric embedded system design paradigm”. NEC Laboratories America. Princeton University. Princeton.

[6] K. Keutzer. “System-level design: Orthogonalization of concerns and platform-based design,” In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Dec. 2000.

[7] L. Cai, D. Gajski. “Transaction Level Modeling: An Overview”. In Proceedings of CODES+ISSS’03, California, USA, October 2003.

[8] S. Pasricha, N. Dutt, M. Ben-Romdhane, “Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration", In Proc of DAC 2004.

[9] S. Pasricha. ”Transaction level modeling of SoC with SystemC 2.0”, In Synopsys User Group Conference, 2002.

[10] W. Klingauf. “Systematic Transaction Level Modeling of Embedded Systems with SystemC”, *Proc. DATE*, 2005.

[11] T. Grötker, S. Liao, G. Martin, S. Swan. “System Design with SystemC”. Kluwer Academic Publishers, 2002.

[12] F. Ghenassia. “Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems”. Springer, 2005.

[13] A. Rose, S. Swan, J. Pierce, J. Fernandez. “Transaction Level Modeling in SystemC”. SystemC TLM whitepaper, 2005

[14] SystemC Verification Library 1.0, 2003, http://www.systemc.org

[15] Transaction Level Modelling Standard 1.0, June 2005, http://www.systemc.org

[16] MiniNoC website. http://www.es.ele.tue.nl/~mininoc/

[17] D. A. Patterson, J. L. Hennessy. “Computer Organization and Design: The Hardware/Software Interface”. 3rd Edition. Morgan Kaufmann Publishers.

[18] V. Reyes, W. Kruijzer, T. Bautista, A. Nuñez. "A SystemC based System-Level Design Tool for Multiprocessor SoC Modeling and Simulation", submitted to ACM Transactions on Embedded Computing Systems, 2006

[19] V. Reyes, W. Kruijzer, T. Bautista, G. Marrero, P. Carballo. 2004. CASSE: A System-Level Modeling and Design-Space Exploration Tool for Multiprocessor Systems-on-Chip. In Proceedings of Euromicro Symposium on Digital System Design, Rennes, France, September 2004.

[20] V. Reyes, W. Kruijzer, T. Bautista, G. Marrero, A. Nuñez. 2005. A Multicast Inter-Task Communication Protocol for Embedded Multiprocessor Systems. In Proceedings of CODES+ISSS’05, New York, USA, October 2005.

[21] W. Kruijzer KRUIJTZER, V. Reyes, W. Gehrke. 2006. Design, Synthesis and Verification of a Smart Imaging Core using SystemC. In Design Automation of Embedded Systems (DAES) journal, Springer, 2006.

[22] van de Wolf, P. de Kock E. Hendrikson T., Kruijtzer, W., Essink, G. “Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach”. 2004. In Proceedings of CODES+ISSS’04, September 2004.

[23] IEEE 1666TM SystemC standard, 2006, http://www.systemc.org

[24] S. Stuijk. “Design and implementation of a JPEG decoder”. Practical Training Report. Technical University of Eindhoven. Eindhoven, The Netherlands. December 2001.

Abreviation

CPU	Central Processing Unit
DSP	Digital Signal Processing
EC	External Component
FPGA	Field Programmable Gate Array
JPEG	Joint Photographic Experts Group
ICCP	Inter-Component Communication Protocol
IDCT	Inverse Discrete Cosine Transform
IP	Intellectual Property
MCU	Minimal Code Unit
MIPS	Million Instructions Per Second
MP-SoC	Multi-Processor System-on-chip
NE	Network Element
NI	Network Interface
NoC	Network on Chip
PE	Processing Element
RAM	Random Access Memory
RTL	Register Transfer Level
SE	Storage Element
SoC	System-on-Chip
TLM	Transaction Level Modeling

Appendix A

A.1 Timing progress

The following graphic illustrates the time spent in each of the tasks that this Master Thesis required. Each task is annotated with the percentage of time spent. In some cases this percentage has been split in the different fields that composed the task, for example the *knowledge acquisition* task which required the study of the *MiniNoC* platform, the CASSE tool, the C++ language and the SystemC library, and the CTAP tool.

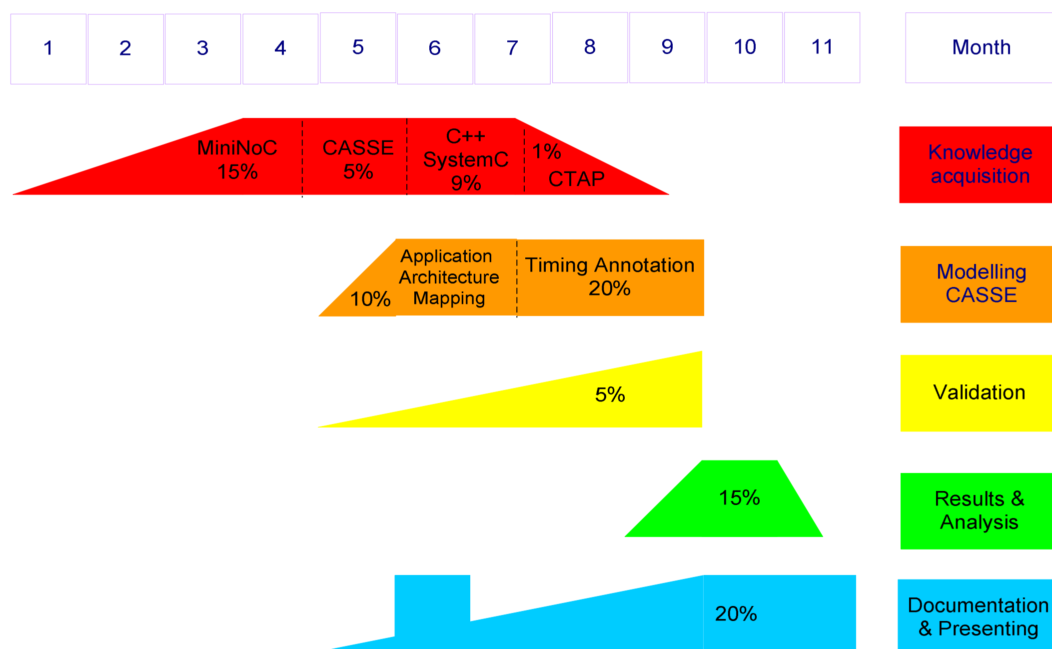


Figure A.1: Timing progress