

Section of Information and Communication Systems (ICS)
Faculty of Electrical Engineering
ICS/ES 815

Practical Training Report

A NUMA Architecture Simulator

J.H.G.M. Vrijnsen
(s461097)

Coaches TU/e:	prof. dr. H. Corporaal, ir. P. Poplavko
Company Coach:	prof. dr. ir. F. Catthoor
Period:	August 26, 2002 – February 21, 2003

Preface

This document presents an overview of an internship for the Electrical Engineering department of the Eindhoven University of Technology, conducted at IMEC, Leuven, Belgium. It is carried out in the scope of the IMEC/Matador¹ project. This project aims at the development of a modular platform-simulation environment to estimate the energy consumption and performance of multi-processor systems in a system-on-chip (SoC) context ([Mar02]), and to verify design methods for such systems.

I would like to thank my supervisor, Henk Corporaal, for providing me the option to work for six months in one of Europe's largest independent research centers. I would also like to thank Peter Poplavko, my daily supervisor, for providing me with lots of help to complete this practical training project. I would like to thank him also for his patience in explaining things to me, and reviewing my practical training report. Furthermore, I would like to thank Twan Basten for taking interest in my work, and for spending his (sparse) free time on having discussions with me about my work. In addition, I would like to thank him for reviewing my practical training report. Finally, I would like to thank Francky Catthoor and Johan Vounckx from IMEC for allowing me to perform my practical training project at IMEC.

¹ Management of TAsks with Dynamic, Overlapping-date and Real-time behavior (MATADOR)

Abstract

In the near future low-cost, portable consumer devices that integrate multi-media and wireless technology will drive the silicon market. The general believe is, that to meet the system requirements of these types of applications, they need to be embedded onto (heterogeneous) multi-processor platforms. For this, a systematic design method and tool support is a must. Furthermore, in order to be able to test the effectiveness of a design method, and/or estimate the energy consumption and performance of a platform for a given application, a simulation environment is needed.

In this report, the implementation of a simulation environment for a multi-processor architecture is described. In order to come to an implementation, a classification of multi-processor architectures has been made. Following from this classification, the NUMA architecture is chosen as starting point for the implementation of the simulation environment. Details of the implementation are presented, as well as results from a case study performed with the simulation environment. This case study consists of the mapping of a JPEG application onto the NUMA architecture. The results of the case study show, that the implemented simulator can be used to measure the performance of the architecture (in terms of cycles) and to obtain timing characteristics for an application. Furthermore, it shows that running an application on a multi-processor architecture based on a NoS could be a feasible solution, however, a lot more work is required to really “reveal” it.

Keywords: multi-processor architectures, classification, shared-memory, SoC, NoS, process networks, simulation, JPEG

Table of Contents

PREFACE	i
ABSTRACT	iii
1 INTRODUCTION	1
1.1 PROBLEM STATEMENT	2
1.2 OVERVIEW	3
2 MULTI-PROCESSOR ARCHITECTURES	5
2.1 CLASSIFYING MULTI-PROCESSOR ARCHITECTURES	5
2.2 DISTRIBUTED MEMORY PLATFORM BASED ON NoS.....	6
2.3 SHARED MEMORY PLATFORM BASED ON NoS.....	7
2.4 SOFTWARE-CONTROLLED SHARED MEMORY HIERARCHY PLATFORM.....	7
2.5 COHERENCY AND CONSISTENCY	8
2.5.1 <i>Cache Coherency</i>	8
2.5.2 <i>Memory Consistency</i>	10
3 NUMA ARCHITECTURE STRUCTURE	13
3.1 REQUIREMENTS FOR THE SIMULATOR.....	13
3.2 ARCHITECTURE STRUCTURE.....	13
3.3 MAPPING OF KPNS ON NUMA ARCHITECTURE.....	16
4 IMPLEMENTATION OF SIMULATOR	19
4.1 INTEGRATION OF ISS INTO SIMULATION BACKEND	19
4.2 SIMULATOR OVERVIEW.....	20
4.3 INTEGRATION WITH ÆTHEREAL	22
4.3.1 <i>Æthereal Packet Format</i>	22
4.3.2 <i>Integration Assumptions</i>	22
4.4 TUNING THE ARCHITECTURE	24
5 CASE STUDY	25
5.1 GOALS OF THE CASE STUDY	25
5.2 JPEG DECODING.....	25
5.2.1 <i>JPEG Process Network</i>	26
5.3 EXPERIMENTS	27
5.3.1 <i>Experiment 1 - Impact of NoS</i>	27
5.3.2 <i>Experiment 2 - Communication vs. Computation</i>	28
5.3.3 <i>Experiment 3 - Latency and Throughput</i>	29
5.3.4 <i>Experiment 4 - Three Parallel JPEGs</i>	31
6 RELATED WORK	33
7 CONCLUSIONS AND FUTURE WORK	35
BIBLIOGRAPHY	37
LIST OF FIGURES	39
LIST OF TABLES	39
APPENDIX A	41

1 Introduction

The main reason to setup a simulation environment for a multi-processor platform is the belief that in the near future low-cost, portable consumer devices that integrate multi-media and wireless technology will drive the silicon market. This will put stringent constraints on the degree of integration (i.e., chip area: all functionality must fit on only a few mm² of silicon) and on their power consumption (0.1-2W). Furthermore, this type of applications will require a computational performance in the order of 1-40 GOPS. Additionally, most of these applications will have stringent real-time constraints and dynamism, e.g., due to user interaction; in addition, they will require programmability of (parts of) the core. All these requirements complicate their implementation considerably. Although current PCs (almost) meet the performance requirement, their power consumption is orders of magnitude too high (10-100W). This renders PCs infeasible for this type of embedded applications, and therefore we want to look at heterogeneous multi-processor platforms. The general belief is, that with these type of platforms, a high performance at a low energy cost is obtained through the exploitation of different types of parallelism in an application, since this parallelism allows to distribute the computational load of an application over several processing elements ([Cha99], [Man00]). Note, that with a multi-processor platform we do not intend to address a multi-chip platform like the Cray T3D, but rather a system-on-chip (SoC): multiple processors, memories and interconnect on a single chip.

In Figure 1-1, an example of a multi-processor platform is shown. Such a platform typically consists of architecture nodes connected to a scalable interconnection network. Each of those

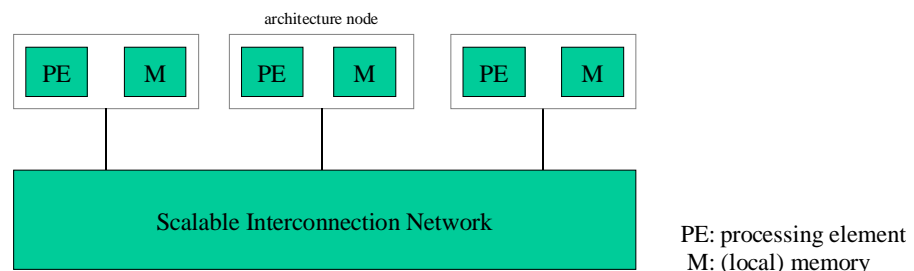


Figure 1-1: Example of multi-processor platform

architecture nodes can contain a processing element (PE) and a (shared or private) memory (M); however, they do not necessarily contain both. A processing element consists of a processor (e.g., a VLIW-like, a RISC-like, a MIPS-like or a Trimedia), and some private (L0) memory, like a cache, register files or even some built-in RAM. The architecture nodes are connected by means of a scalable interconnection network, e.g., a network-on-silicon (NoS), a switching network or even a simple global bus (e.g., in case of a two-processor system).

The design of such multi-processor platforms is a complex task. Therefore, it is necessary to develop a systematic design method and tool support to embed multi-media applications on multi-processor platforms. In order to be able to test the effectiveness of a design method, and/or estimate the energy consumption and performance of a platform, a simulation environment is needed.

1.1 Problem Statement

Some work on implementing a simulation environment for simulation and verification of the systems mentioned in the previous section has already been done by IMEC and Philips, in the Matador and Æthereal projects, respectively. Results from these projects served as input for this practical training project; therefore, first a brief overview of their achievements is presented.

The simulation environment that is under construction (and partially operational) in the Matador project ([Yan01]) supports the following:

- processor models based on instruction-set simulators (ISS), featuring energy-consumption models;
- performance/energy models for instruction- and data-memory hierarchies;
- simulation control that implements a timing model and synchronizes the execution of multiple processor models;
- main shared and intermediate memory models that can be accessed directly by cache controllers or processor nodes;
- communication between processors based on UNIX shared memory communication.

One of the current goals of the Matador project is to extend the simulation environment to support transaction-level communication based on segmented bus and/or network-on-silicon (NoS) protocols.

The Æthereal project focuses on the design of multi-hop networks-on-silicon up to the transaction-level ([Rij01]). A NoS consists of routers and network interfaces, and can basically be seen as an “on-chip Internet”: architecture nodes are connected to the NoS by means of network interfaces, and messages from a source node to a destination node pass through several routers, in a (more or less) similar way as computers communicate with each other via the Internet.

The goal of the Æthereal project is, to provide the system designer with the services at the transaction-level and the figures for NoS area, performance and power as a basis for system design. For this purpose, among other things, Philips is developing a simulation environment for its NoS; this simulation environment is based on a SystemC library and supports the following ([Pop02]):

- a router module that corresponds to a subset of a router specification implemented in hardware at Philips;
- a network interface (NI) module that supports communication via FIFO channels of the YAPI² library. A more advanced NI supporting a transaction language is currently under development;
- a network model builder, that allows specifying an arbitrary network topology. The builder plugs router- and network interface modules into the network model;
- static configuration of communication channels specified by the user. It includes the mapping of FIFO channels to a network path and the mapping of process ports to network interfaces.

The aim of this practical training project is to build a simulation environment for a multi-processor architecture with transaction-level communication based on a NoS protocol. For this, a classification of multi-processor architectures must be made, in order to select one of these architectures as starting point for the simulation environment. Furthermore, a case study must be performed, in which a driver application must be mapped onto the chosen multi-processor architecture, and simulated with the simulation environment. The goal of this case study is, first of all, to test the functional correctness of the simulator; secondly, it allows us to perform an architecture exploration, showing the impact of architecture parameters (e.g., processor speed, memory size) on the performance of the application.

² YAPI (Y-chart Application Programmer’s Interface) is a parallel programming interface meant for functional modeling of hardware/software blocks communicating through FIFO-channels (e.g., Kahn process networks; [Koc02])

1.2 Overview

In order to come to an implementation of a simulation environment for a multi-processor platform with transaction-level communication, it is important to have some insight in multi-processor architecture options that could be of interest. A classification is presented in Chapter 2, together with some notes about cache coherency and memory consistency, since these are important issues concerning multi-processor architectures.

Based on this classification and the constraint of using the *Æ*thereal simulation environment, the shared memory platform connected through a NoS (or NUMA³ architecture) is chosen as the target architecture for the simulator. In Chapter 3, a more detailed overview of all (relevant) hardware components of this architecture is presented, while in Chapter 4 the most crucial simulator implementation issues are presented.

The case study, consisting of the mapping of a JPEG application onto the NUMA architecture, is described in Chapter 5, together with the results of this case study.

Chapter 6 gives a small overview of other multi-processor simulators. For this practical training project, no real attention was paid to other simulation environments, based on the constraints; however, for completeness, this small overview is added.

Finally, Chapter 7 presents conclusions and hints for future work.

³ NUMA: non-uniform memory access, explained in Chapter 2

2 Multi-Processor Architectures

A brief introduction to multi-processor platforms has been provided in the previous chapter. In this chapter, we present a classification for such platforms. We do however not discuss all possible architectures, but we restrict ourselves to those options of most interest for our work.

First, in Section 2.1, two “axes” for the classification of multi-processor architectures are presented. Based on these axes, Sections 2.2, 2.3 and 2.4 discuss three different types of architectures. Finally, in Section 2.5, two important issues concerning multi-processor architectures are presented: cache coherency and memory consistency. Note, that we introduce coherency here since it is of crucial importance in multi-processor systems, but as we will see in the next chapter, the implementation of a cache coherency protocol for our simulator is beyond the scope of this practical training project.

2.1 Classifying Multi-Processor Architectures

In this section, two axes for the classification of multi-processor architectures are presented: *memory architecture* and *interconnection network*.

The memory architecture forms one of the most crucial components of multi-processor platforms, since, first of all, the memory architecture needs to guarantee a sufficiently high throughput of data such that timing constraints can be met. Secondly, the energy consumption of the memory architecture should be kept as low as possible, to ensure the autonomy of the (battery-powered) embedded devices. Finally, the area used by on-chip memories should be minimized, since even on current single-processor architectures memory dominates the on-chip surface. Therefore, it is fair to classify multi-processor architectures based on the memory architecture they use; the most common are *shared memory* and *distributed memory* architectures. Note, that a memory architecture can be either *physically* shared or distributed, or *logically* shared or distributed; however, we only look at the logically options.

In practice, all memories are physically distributed; the only case in which one does not have a distributed memory is, when in Figure 1-1 one of the architecture nodes would consist of a memory only and the other nodes of processing elements only. According to our classification, distributed memory refers to the fact that a memory is local to a processing element (i.e., in the same architecture node) and visible only to this processing element. In other words, that processing element has a private address space, and read/write access to this private address space only.

On the other hand, a shared memory is accessible by multiple processing elements: there is a global address space. Note, that the non-distributed memory option, mentioned before, is fully shared: there is only one central memory, which is accessible by all processing elements, and thus shared by all processing elements. Next to that, an architecture is fully distributed, when all architecture nodes contain local memories (not shared with other processing elements). Obviously, mixed architectures of both distributed- and shared memory exist (e.g., clusters of shared memory).

In order to meet the throughput and energy consumption constraints, one could consider the use of caches. Caches are small memories, in which contents from a memory are stored that are accessed often by a processing element. A cache can also be used for example to pre-load blocks of memory from the main memory, so that the processor does not need to stall when performing a load of sequential memory locations. Usually, caches have faster access times and lower energy consumption than “normal” memories. Two types of caching are possible. In Figure 2-1a, only data from the local memory is cached: the memory management unit (MMU) directs loads and stores of its corresponding processor either to the local cache or to the NoS for a remote memory access. In Figure 2-1b, data from both local and remote memories are cached. The first type of cache can exist in both distributed- and shared memory architectures, the latter only in shared memory architectures.

Another crucial component of a multi-processor architecture is the interconnection network used to “glue” the architecture nodes together. The main options for this interconnect are *router-based NoS* and *memory hierarchy*. The network-on-silicon (NoS) has already been briefly introduced in the previous chapter. The memory hierarchy is basically a shared memory architecture, where all processing elements are connected to each other through several layers of (shared) memory.

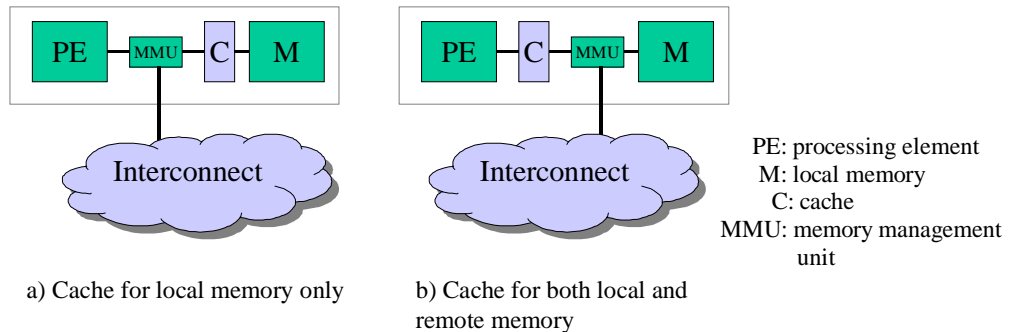


Figure 2-1: Possible locations of cache

Based on the two axes presented for classification, memory architecture and interconnection network, we present in the next sections the following multi-processor architectures:

- a distributed memory platform based on NoS;
- a shared memory platform based on NoS;
- a (software-controlled) shared memory hierarchy platform.

2.2 Distributed Memory Platform based on NoS

A general overview of the distributed memory platform is presented in Figure 2-2: it consists of multiple architecture nodes, containing both a processing element and a memory, connected to the network interfaces (NI) of the NoS. All memories in the system are private, and visible only in their architecture nodes (there is only a local, thus private, address space). When for example the processing element of architecture node 1 in Figure 2-2 needs data from the processing element of node 3, explicit communication has to be used. Message passing is used for this communication: blocks of data are transferred from one node to another, if needed.

Although not shown in Figure 2-2, the distributed memory platform can also contain caches. However, this platform can only cache data from the local memory (Figure 2-1a), since the processing elements have only access to their own local address space.

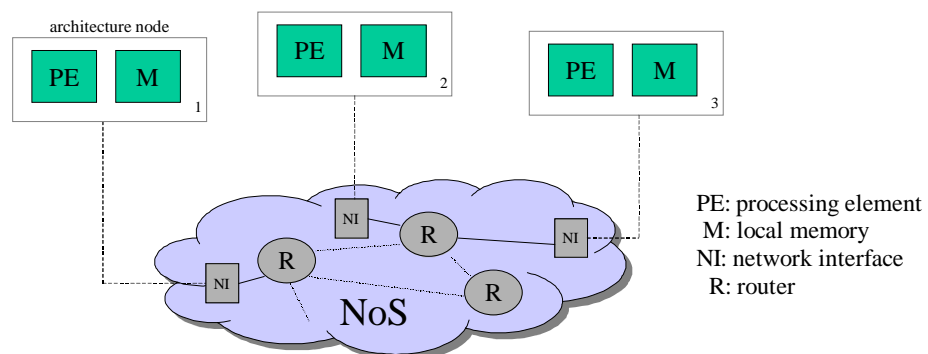


Figure 2-2: General overview of the distributed memory platform

The main advantage of this architecture is, that it is easy to scale to larger applications in comparison to other architectures, like for example the shared memory architecture presented in the following section. A major disadvantage is, that the partitioning of tasks over the processing elements

should be done carefully to avoid unnecessary communication, since the explicit data communication between processors is much more expensive -especially in terms of energy consumption- than fetching local data. In addition, the performance and latency can suffer from this explicit data communication; however, these can largely be “hidden” in a well-designed distributed memory system. Another disadvantage is the fact that the dimensioning of the local memories is important: they have to be sufficiently large to be able to contain all local data.

2.3 Shared Memory Platform based on NoS

Another option for a multi-processor platform is the shared memory platform: all the memories in Figure 2-2 are accessible by all processing elements. In other words, there is a global address space. Communication between processing elements can occur implicitly as a result of conventional memory access instructions (i.e., loads and stores). In this so-called *non-uniform memory access* (NUMA) approach ([Cul99]), the local memory controller determines -based on the global address- whether to perform a local memory access or a message transaction via the NoS with a remote memory controller. These accesses are called non-uniform, since an access to a local memory location is performed faster than an access to a remote memory location, due to the delay introduced by the interconnection network (NoS in Figure 2-2). With this type of architecture, it is also possible to have architecture nodes consisting of a processing element or memory only. Take for example the non-distributed option presented in the introduction: one architecture node consisting of a memory only, and several architecture nodes consisting of a processing element only. Furthermore, it is possible to have a partially shared address space instead of a globally shared address space. E.g., it is possible that in Figure 2-2 architecture nodes 1 and 3 share their address space, and nodes 2 and 3. In that case, node 2 has no direct access to the memory of node 1. If communication is needed between node 1 and 2, message passing has to be used.

As mentioned in the introduction, caches can be used to improve the performance of the architecture. Both types of caches presented in Figure 2-1 are applicable to a shared memory platform. However, care has to be taken when option b of Figure 2-1 (caching of both local and remote memories) is used, since in that case, various copies of a variable can exist in various caches at a certain point in time. When one of these copies is changed, it has to be ensured that all other copies are also changed, or at least are invalidated. For this, a cache coherency protocol is needed. Cache coherency is discussed in Section 2.5, together with memory consistency -the order in which accesses to variables in a shared memory become visible to other processors- since this is also an important issue concerning shared memory architectures (if they allow duplication of “life” data).

The main advantage of a shared memory architecture is, that the dimensioning of memories is less crucial, because it is possible to store data in memories that are not local to a processor, due to the global address space. A major disadvantage is the fact that cache coherency and memory consistency have to be taken care of, which will have an impact on the complexity of the design, as well as on the performance and energy consumption of the architecture.

2.4 Software-Controlled Shared Memory Hierarchy Platform

Figure 2-3 shows an example of a multi-processor architecture based on the software-controlled shared memory hierarchy design paradigm. This is the approach taken by the Matador project in IMEC ([Mar02]).

The memory hierarchy of the architecture presented in Figure 2-3 consists of several layers. Memories in each layer are shared, partially shared or privately owned by the processing elements. The connections between several layers of memory and between memories and PEs are bus-based.

The caches and software-controlled memories (SRAM’s or scratchpad memories) can be multi-ported. The term “software-controlled” means that, in contrast to for example hardware-controlled caches, it is the compiler’s job to program loading of frequently accessed data structures from the

higher level memory (e.g., from the off-chip main memory), to a lower level memory (e.g., a scratchpad). Special units, like DMA-units, can do the actual loading to offload the PEs.

The main advantage of this architecture is the fact that (on-chip) area can be saved by sharing the memories at higher layers of the hierarchy. However, at the lower layers, more distribution will be needed to guarantee the performance while still keeping the energy consumption low. A major potential for exploration, but at the same time a major disadvantage for design complexity, is the fact that multiple processor-memory paths are possible (e.g., in Figure 2-3, PE2 can access the main memory via the L1-DCache, or via the small shared memory on its right), allowing the interconnect to reorder many access to the memories. This means, that also for this type of architecture memory consistency has to be taken care of, which will be done by the software control based on compiler decisions ([Mar02]).

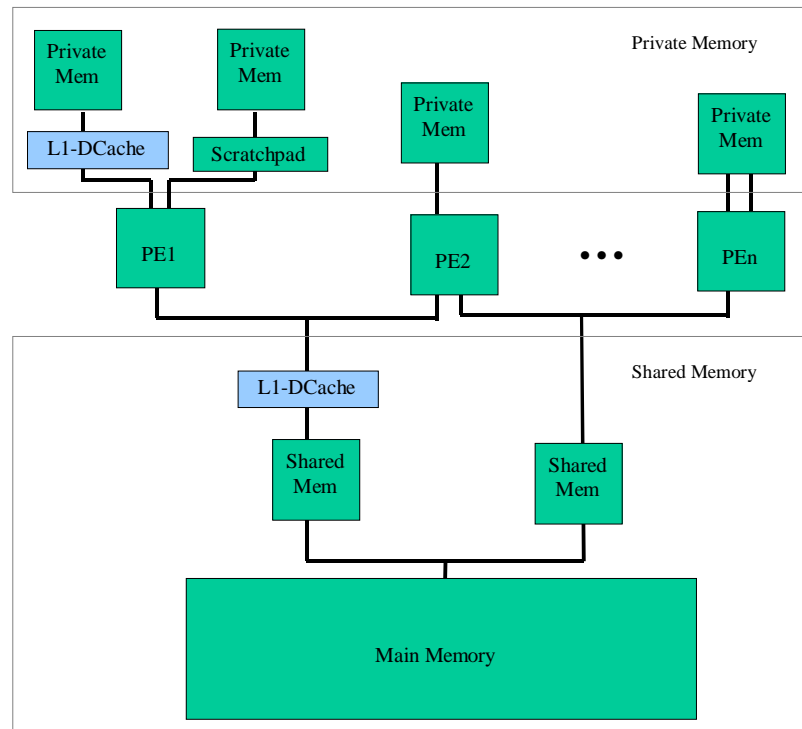


Figure 2-3: Typical configuration of a shared memory hierarchy (taken from [Mar02])

2.5 Coherency and Consistency

Cache coherency is the management of a cache so that no data is lost, corrupted or overwritten. It is the topic of Section 2.5.1. A memory consistency model specifies constraints on the order in which memory operations (to the same or to different locations) can appear to execute, with respect to each other. In Section 2.5.2, memory consistency is explained starting from the coherency problem. Note, that consistency and coherency are two orthogonal subjects, although the coherency problem is aggravated in the presence of caches.

2.5.1 Cache Coherency

The concept of a cache has already briefly been introduced in Section 2.1. Here we discuss the problems that arise when using caches in a shared memory environment and how to deal with them. However, first a short explanation of how caches actually work is presented. To simplify matters, we start with assuming that we have a single memory with a single cache and a single processor.

Basically, there are two approaches for designing caches. The easier type of cache is a *write-through* cache: when a processor performs a write to its memory, the relevant cache entry is updated

and the cache issues a write to the main memory. In this way, cache coherency is maintained, i.e., the main memory accurately reflects the contents of the cache.

The main disadvantage of this approach is, that every time a processor writes to its cache, also a write to the main memory is issued, which is inefficient and costly. Issuing the write to the main memory once the processor has definitely finished with that cache location can solve this problem. This is what a *write-back* cache tries to do: it writes out the cache location to the main memory sometime after the actual write took place, depending on when the caching algorithm decides to do so. It is important to note that for the period between the write to the cache and the write-back to the main memory, that memory location is not coherent with the cache.

Now consider the shared memory situation presented in Figure 2-4a: both processors P_1 and P_2 are using the variable x from the main memory for their calculations. At some point in time, P_1 updates the value of x . For a write-through cache (Figure 2-4b), the updated variable is immediately written back to the shared memory; however, the contents of the cache of P_2 are not updated, and thus the contents of P_2 's cache are not coherent with that of the shared memory and P_1 's cache. In case of a write-back cache (Figure 2-4c), it becomes even worse: the update of P_1 is not immediately written back to the shared memory, so both the shared memory and P_2 's cache are not coherent anymore.

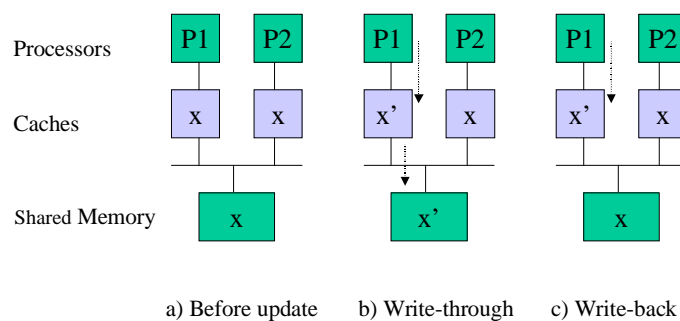


Figure 2-4: Sharing of data in a multi-processor environment

The solution to maintaining the coherency of caches is a cache coherency protocol, implemented in special support hardware. The two main protocol types are presented in Figure 2-5: the write-invalidate (Figure 2-5b) and the write-update protocol (Figure 2-5c). For the first protocol, an update to a memory location invalidates that location in all other caches, while for the latter protocol, this update causes an update for that location in all other caches.

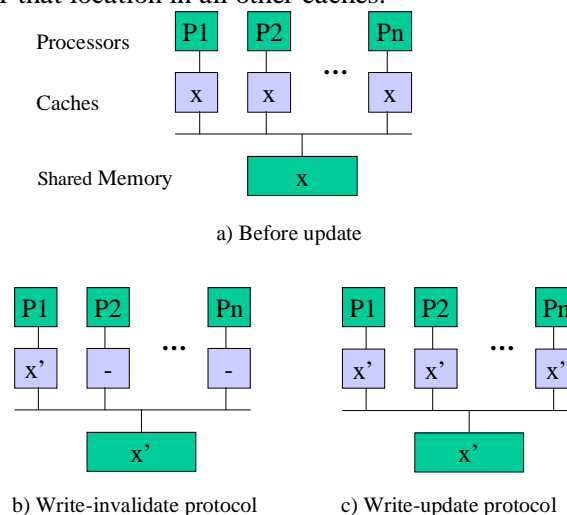


Figure 2-5: Cache coherency protocols (shown for write-through cache)

Obviously, implementing these protocols in hardware is non-trivial (and beyond the scope of this practical training project). A more detailed treatment of caches and cache coherency can be found in [Cu199].

2.5.2 Memory Consistency

Coherency is essential if information is to be transferred between processors where one processor writes to a location, that the other reads. Eventually, the value written will become visible to the reader. However, coherency says nothing about **when** the value written will become visible to other processors. Often in writing a parallel program, we want to ensure, that a read returns the value of a particular write; that is, we want to establish an order between a write and a read. Consider for example the code fragments executed by processors P_1 and P_2 in Figure 2-6. It is clear, that the programmer intends for processor P_2 to spin idly until the value of the shared variable `flag` changes to 1 and then to print the value of variable `A` as 1, since the value of `A` was updated before that of `flag` by processor P_1 . So, the programmer assumes, that the write of `A` becomes visible to P_2 before the write of `flag`, and that the read of `flag` that breaks processor P_2 out of its `while`-loop completes before its read of `A` (a `print`-statement is essentially a read). However, in the shown situation, where `A` and `flag` are located in different (shared) memories, it is possible that `A` will be printed as 0 by processor P_2 . If for example P_1 does not wait for some acknowledgement that the write of `A` has finished, but immediately writes `flag=1`, it is possible that P_2 prints `A=0`, if the write to `A` is slow compared to the write of `flag`.

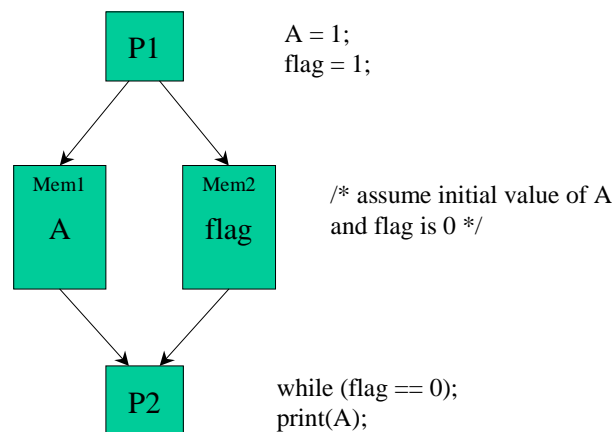


Figure 2-6: Example of consistency problems in a multi-processor environment: intuitively, `A` should be printed as 1. However, without some constraints on the ordering of memory accesses, this does not have to be the case

Clearly, if no order between memory accesses is specified, the programmer is not sure what the outcome of its program will be. Memory consistency models are the solution to this problem, because they specify constraints on the order in which memory operations must be performed with respect to one another (in other words, to become visible to all processors). Several consistency models exist; one of the simplest and most intuitive models is the *sequential consistency* model. According to [Cul99], three conditions suffice to guarantee sequential consistency in a multi-processor platform:

1. no reordering of memory operations from the same processor is allowed;
2. after a write operation is issued, the issuing processor waits for the write to complete before issuing its next operation;
3. no read operation can get a variable written by a write operation, if this write operation is still busy updating all the copies of the given variable (e.g., in caches).

Looking at Figure 2-6, these conditions ensure that the write of `A` is not issued before the write of `flag` (condition 1), and that the write of `flag` is only issued after the write of `A` has finished (condition 2). Furthermore, P_2 will only be able to read the contents of `flag` once the corresponding write of P_1 has completed, and all other copies of `flag` have been updated (condition 3; however, in this example there are no other copies). Thus the outcome of the `print`-statement will be 1, as was intended.

More relaxed consistency models exist (e.g., preserving write-to-write or write-to-read order only); however, we do not discuss them here (see for example [Cul99] for more information about other consistency models).

3 NUMA Architecture Structure

In this chapter, we present a functional description of the hardware components of the NUMA architecture, since this architecture is chosen as starting point for our simulation environment. Furthermore, we describe how we map applications onto this architecture. However, first an overview of the requirements for the simulation environment is presented.

3.1 Requirements for the Simulator

As presented in Chapter 1, the aim of the practical training project is to build a simulation environment for a multi-processor architecture. This environment should obey the following requirements:

- functional:
 - perform cycle based simulation for the processor model and transaction based simulation for other parts of the simulator;
 - provide performance metrics;
 - provide modular plug-in of hardware blocks;
- non-functional:
 - use the shared memory and processor models of the Matador simulation environment;
 - use the (cycle-accurate) *Æ*thereal simulation environment to simulate a NoS.

Finally, since the central principle of the *Æ*thereal simulation environment is the simulation of YAPI communication via a packet-switched network having an arbitrary topology ([Pop02]) and since it is based on the SystemC library, our simulation environment should obey the following secondary requirements:

- use SystemC as simulation backend;
- use YAPI to model network connections.

Based on these requirements, the NUMA architecture based on NoS from the previous chapter is an obvious choice for our simulation environment, since this architecture is based on a NoS and is a shared memory platform. In the following section we present a functional description of this architecture.

3.2 Architecture Structure

In Figure 3-1, a more detailed overview of the shared memory platform that we took as starting point for our simulation environment is presented. Shown in this figure are two architecture nodes, connected through a NoS.

Since we are looking at heterogeneous multi-processor platforms, the processor P_x of every architecture node can be of any type, e.g., a VLIW-like, a MIPS-like or a Trimedia. However, the Matador environment uses the ARM as processor, so in our simulator the processors also consist of ARMs. Every processor has its own private data- and instruction memory (M), in which processes can be loaded. Next to that, every architecture node contains a shared memory M_x , which is accessible to every other processor. We assume that only the private memories are cacheable, since, in order to allow caching of shared memories, a cache coherency protocol has to be implemented, which is beyond the scope of this practical training project. In addition, as we will see in Section 3.3, in the application benchmarks that we use, we have chosen to use “read FIFO” and “write FIFO” statements that copy data between private and shared memories, and therefore caching of shared memories would

not make sense. Next to that, we assume that every node contains both a processor and a shared memory. Processors can perform load, store or swap operations on these memories. A swap is an atomic operation, used for synchronization of accesses to shared memory locations. It consists of an indivisible exchange operation (load followed by store) that interchanges a value in a register from Px for a value in the shared memory.

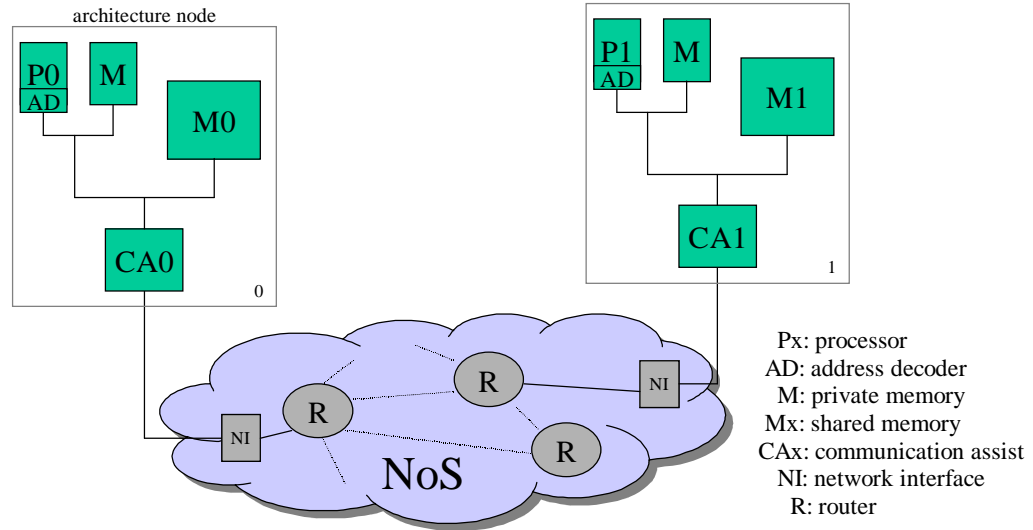


Figure 3-1: NUMA architecture structure

In order to be able to access all shared memory locations, a global shared address space is used, as shown in Figure 3-2. Every processor has a local (virtual) address space, containing only those parts of the global address space that are accessible by this processor. The address range of every shared memory Mx is mapped onto a small piece of the total shared address space. For this, the following convention is used: the k most significant bits of an address in the shared address range determine the physical location (node number) of the address. In this way, we are able to address 2^k shared memories, where the size of each shared memory is the size of the total physical shared memory divided by the total number of shared memories.

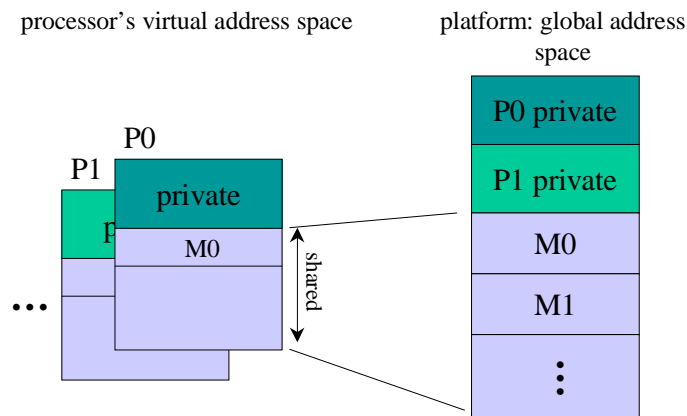


Figure 3-2: Memory model for NUMA architecture

What happens if for example P0 in Figure 3-1 wants to perform a load of a memory location in a shared memory? The address decoder of P0 translates the virtual address into a global physical address that is presented to the memory system. If this physical address is local to processor P0, it will access M0 via the local bus, and M0 will simply respond with the contents of the desired location. If however the physical address is not local to P0, a network transaction with a remote architecture node has to be issued. The communication assist, as illustrated in Figure 3-3, will perform this access to a remote

memory location. Therefore, the local communication assist (CA0) appears as a "pseudo-memory" to the processor while it accesses the remote location. This "pseudo-memory" controller accepts the load transaction from the processor (1 in Figure 3-3), extracts the node number from the global physical address (node 1 for example) and issues a network transaction to the remote node to access the desired memory location (2). The remote CA (CA1) receives the network transaction, loads the desired memory address (3) and issues a response transaction with the loaded data to the source node (4). CA1 appears as a "pseudo-processor" to the memory system on its node when it issues the load to the memory system (M1). Eventually, the response transaction will arrive at the originating pseudo-memory controller (CA0). This controller will then complete the load operation as if it were a (slow) memory module (5) with respect to P0.

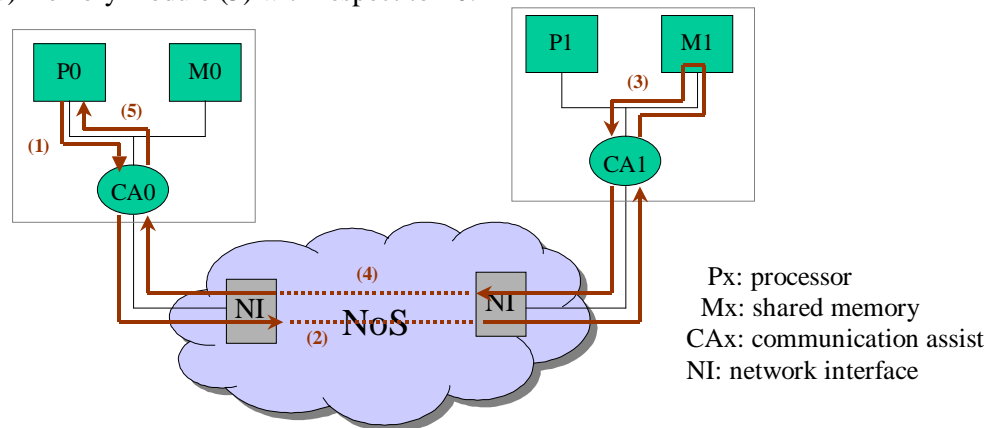


Figure 3-3: Issuing a load to a remote memory location

Following from this example, the communication assist can be split into two separate units: a pseudo-memory and a pseudo-processor unit. The pseudo-memory unit acts as a memory to the processor that wants to access a remote memory address, and provides the following functionality:

- it accepts the memory request type (load, store or swap), size of the memory access (word, half word, byte), address to access, data to store (in case of a store access) and a memory lock signal (in case of a swap operation);
- it issues a network transaction to the remote memory location (pseudo-processor unit of that node) with type of access, address, size of the memory access, data and lock;
- it accepts the response of the remote location (pseudo-processor unit);
- it returns the requested data to the processor (in case of a load).

The pseudo-processor unit, on the other hand, acts as a processor to a shared memory: it accesses this memory on behalf of a remote processor, as if it were a real processor itself. This unit provides the following functionality:

- it accepts a network transaction from a pseudo-memory unit and translates this into a bus-transaction for the shared memory;
- in case of a load operation, it accepts the response of the shared memory and translates this response to a network transaction for the requesting pseudo-memory unit;
- if the current request of a pseudo-memory unit is part of a swap, it will continue serving this pseudo-memory until the swap has finished, before serving any other request.

Note, that in the situation shown in Figure 3-1 deadlock can occur, if for example processor P0 wants to issue a memory access to M1 at the same time as processor P1 wants to issue a memory access to M0. A simple local bus supporting one outstanding task is inadequate to avoid this deadlock: both P0 and P1 will be stalled in their remote memory accesses, since they both block their local bus for the access of the other processor. Therefore, we assume that the local bus is decoupled from the accesses to the remote memories by means of a separate bus from address decoder to the pseudo-memory module. In that case, if the processor wants to access a local memory location, the local bus is used, whereas for a remote memory location this latter bus is used, leaving the local bus free for an access from a remote processor.

The (topology of the) NoS shown in Figure 3-1 can be described by an undirected graph consisting of nodes and links. An example of a topology is presented in Figure 3-4; arbitrary topologies are possible. The nodes consist of routers (R) and network interfaces (NI). They are connected to links via ports (not shown in the figure). Every port and every link is full duplex; one flit of data is transmitted in all links (in all directions) every clock cycle. A *flit* is the smallest physical unit of information that can be transferred across a link.

The NoS is a packet switching network ([Rij01]), since the current \AA ethereal environment only supports packet switching. This means, that transaction data coming from an architecture node has to be segmented in packets, to be able to be sent through the network. The segmentation is performed by means of network interfaces (NI in Figure 3-1). A packet is composed of a header and a payload. The payload is the actual transaction data to be transmitted over the network; the header contains information that is used by the routers (R) to forward the packet to its destination. In a packet switched network, links are not reserved for connections, so the NoS provides a best-effort service. This means that traffic is never lost, even if there is a lot of contention in the network. However, no latency or throughput guarantees are given. Furthermore, we assume that the network is order preserving, i.e., packets from a source are delivered in-order at the destination. From a memory consistency point-of-view, this allows the use of “posted stores” to remote memory locations (stores without waiting for an acknowledgements that it has successfully finished). However, even without this guaranteed order per network path, posted stores can be used in some cases, as is presented in the next section.

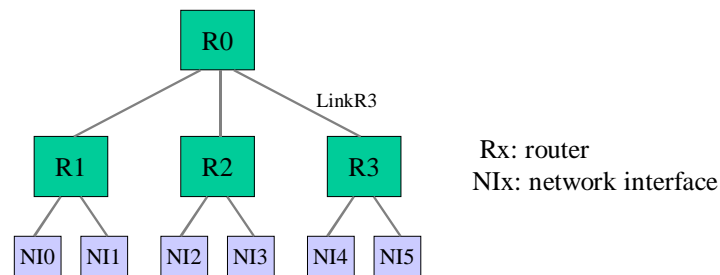


Figure 3-4: Example of network topology

3.3 Mapping of KPNs on NUMA Architecture

We use process networks, which are a derivative of the Kahn Process Network model of computation (KPN, [Kah74]), for application modeling. A process network is a directed graph, where nodes correspond to sequential processes and edges correspond to *FIFOs*, i.e., finite-capacity first-in-first-out buffers. Data that is transported through a FIFO is called a *token*. All sequential processes run in parallel and only communicate and synchronize with each other by reading from or writing to FIFOs. Processes block when reading from an empty FIFO or when writing to a full FIFO. For the modeling of process networks, we use YAPI ([Koc02]): a C++ library with a set of rules, which can be used to model stream processing applications as a process network.

A simple example of a process network is presented in Figure 3-5. This process network consists of a producer, which writes tokens to a FIFO, and a consumer, which reads the tokens from the FIFO. From this example one can conclude, that the main job for the read and write operations is to copy data from a private memory to a shared memory (and vice-versa): a token that is produced in the producer process (present in its private memory) is written to the FIFO buffer (a piece of shared memory). The consumer reads the token from the FIFO and moves this token from the shared memory to its private memory for processing.

As shown in Figure 3-5, access to the FIFOs is controlled by means of semaphores. These semaphores ensure that, if the order between the `acquire-` and `load/store(Token)-statements` is preserved, it is not possible for a process to accidentally access a FIFO that is currently being accessed by an other process, since the process first has to acquire the semaphore before being able to read from or write to the FIFO. Therefore, the order between the `acquire-` and `load/store(Token)-statements` should be preserved, to ensure the necessary memory consistency.

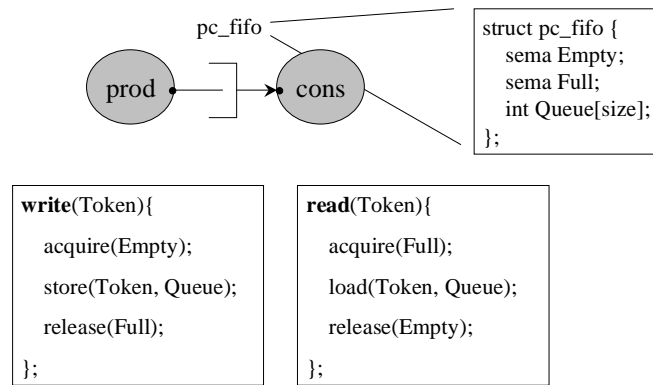


Figure 3-5: Example of process network with a 1-entry FIFO buffer

Figure 3-6 presents the mapping of the presented producer-consumer example onto the NUMA architecture. In general, it is possible to map multiple processes onto one architecture node, depending on the process network. However, a static schedule has to be made for this, or, to allow dynamic scheduling of multiple processes on one processor, an operating system is needed; however, both are beyond the scope of this practical training project and therefore we assume the mapping of one process per processor. As already mentioned in the previous section, we want to use “posted” stores: stores without waiting for an acknowledgement that they have successfully finished. The advantage of using posted stores is, that they are less expensive to issue through the network with respect to “normal” stores, where you wait for an acknowledgement that it has finished, before issuing the next memory access. The advantage of the normal stores is the fact that memory consistency is implicitly ensured, due to this waiting for an acknowledgement. However, in the case of posted stores, we are not able to guarantee order preserving of the memory accesses, if we send the posted stores to two or more memories through two or more paths (see Section 2.5.2). Therefore, we map a FIFO into one shared memory, as shown in Figure 3-6: in that way, order preserving (and thus memory consistency) is ensured, because stores to this memory are issued in-order (by the process) and handled in-order (by the memory). Furthermore, since a posted store is less expensive to issue through the network than a load operation and since the majority of a write operation (see Figure 3-4) consists of stores, whereas the majority of a read operation consists of loads, we put the FIFO memory close to the process that consumes (reads) the data from this FIFO. In that way, most of the loads are performed via the local bus.

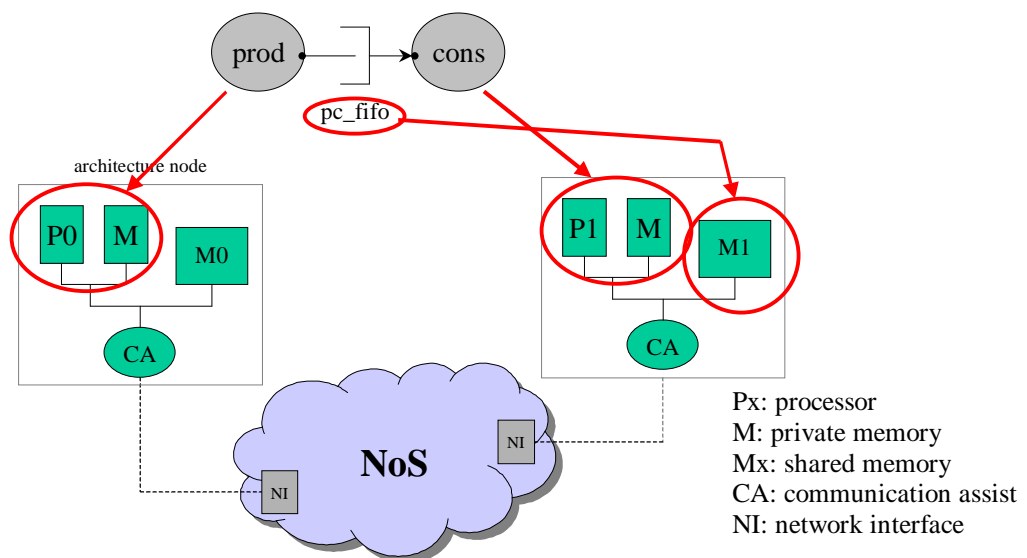


Figure 3-6: Mapping of process network on architecture

Note, that the mapping assumptions presented in this section apply to all process networks, not only to the producer-consumer example.

4 Implementation of Simulator

In this chapter, details of the implementation of a simulation environment for the NUMA architecture are presented.

Since one of the most crucial parts of a multi-processor simulation environment is the processor model, we start in Section 4.1 our discussion with how third-party processor models can be integrated into a simulation backend. In Section 4.2, some implementation details are presented, while in Section 4.3 we discuss how the simulation environment is integrated with the *Æ*thereal environment. Section 4.4, finally, provides an overview of what parameters in the environment can be changed.

4.1 Integration of ISS into Simulation Backend

This section is based on [Ben02].

A general overview of a simulation environment for a multi-processor architecture is shown in Figure 4-1. It consists of processor models, memory models and an interconnection network model, all controlled by and connected to a simulation backend. In this figure, the processor models are shown as separate processes, and the simulation backend, memory models and interconnection network model are shown as one process. The reason for this separation is, that, first of all, processor models are present for different levels of simulation abstraction. Furthermore, several possibilities of integrating a processor model into a simulation backend exist.

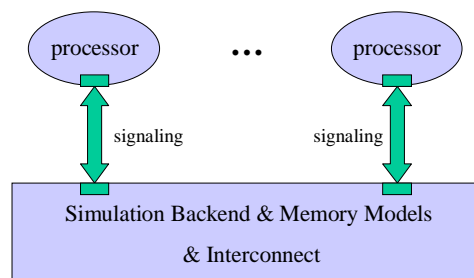


Figure 4-1: General simulation model

In Table 4-1, the most common processor models and their most common ways of integration into a simulation backend are shown. The table columns from left to right correspond to increasing simulation speed and decreasing simulation accuracy, whereas the rows show the signaling method between the different simulation models. For this signaling method, we have included two options: UNIX signaling, which is used in the Matador simulation environment ([Mar02]), and SystemC signaling, since SystemC is the preferred simulation backend (see Section 3.1).

	HDL (cycle accurate model)	ISS (instruction accurate model)	Native processes (transaction level model)
UNIX signaling		remote ISS	X
SystemC signaling		linked ISS	e.g., YAPI

Annotations in the table:
 - An arrow labeled '15x speedup' points from the 'remote ISS' cell to the 'linked ISS' cell.
 - An arrow labeled '10x speedup' points from the 'remote ISS' cell to the 'Native processes' cell.

Table 4-1: Simulation granularity levels

Hardware designers often use hardware-description language (HDL) simulators to validate their work. An example of such a cycle-accurate simulator is the Trimaran VLIW simulator ([TRI]). However, since HDL simulators model their micro-architecture in too much detail, they are expected to be inefficient for simulating complex processor cores.

Less accurate simulation of a micro-architecture is provided by means of an instruction-set simulator (ISS), which efficiently tracks software execution. An example of an ISS is the ARMulator ([ARM]). Embedding such a simulator in a simulation backend is possible in two ways, as shown in Table 4-1: using UNIX inter-process communication (IPC) and wrappers, or linking the ISS directly into the simulation backend. In the first case, the ISS and the simulation backend run as distinct processes on the host system, and they communicate via inter-process communication primitives. The wrapper (green block in Figure 4-1) realizes the interface between the ISS and the simulator. It has two key functions: it ensures synchronization between the system simulation and the ISS, and at the same time it translates the information coming from the ISS into cycle-accurate bus-transactions that are exposed to the rest of the system. This is a very effective way of integration, when the communication between the ISS and the rest of the system is sparse in time. Unfortunately, when the interaction between processor and the simulation backend is tight, IPC may become a serious simulation speed bottleneck. According to [Ben02], this “*remote ISS*” approach has approximately a 10 times slower execution speed than using the “*linked ISS*” approach (integration of the ISS into the simulation backend, as one process). For this second approach, the ISS must be modified to support the inter-process primitives defined by the simulation backend. As shown in the table, according to [Ben02] this method gives a 15 times speedup compared to an HDL model using UNIX signaling.

A final option shown in the table is to eliminate processor models completely and run the application code as native processes, e.g., using YAPI models. A major drawback of this method, however, is less simulation accuracy in terms of timing.

The processor model used in the Matador simulation environment is the ARMulator, an ISS for the ARM core ([ARM]). This leaves only the remote- and linked ISS options open for our multi-processor simulator. However, since we do not have the full source code of the ARMulator, we have (for now) not been able to integrate this ISS into the SystemC simulation backend. Therefore, we have chosen to use the remote ISS approach, probably paying the price of reduced simulation speed.

4.2 Simulator Overview

A global overview of the developed simulation environment for the NUMA architecture is presented in Figure 4-2. Shown in this figure is the situation for one architecture node.

The ISS, processor model and the model for the private memory of the core run as a separate UNIX process and are taken from the Matador environment. In order to integrate the ISS model into the SystemC simulation environment, the Matador processor model had to be modified, since it uses UNIX shared memory as memory model and semaphores to control access to this shared memory. However, we want to use SystemC as simulation backend, with a shared memory model inside SystemC. Therefore, Matador’s processor model has been modified to communicate via a UNIX pipe with a so-called synchronization module in SystemC. Such a synchronization module, as shown in Figure 4-2 (`sync`), is instantiated for each ISS. It synchronizes the shared memory accesses of the processor to the global timing model of SystemC and is therefore only needed for simulation purposes; it is not a model for a hardware block. In other words, this synchronization module represents the processor model to the SystemC simulation backend, since it receives a memory request from the processor, waits for the synchronization time and then passes the processor’s memory request to the address decoder module, as if it were the processor itself. The implemented synchronization module provides the following functionality:

- it accepts load/store requests from the processor;
- it receives the current processing time of the ISS from this ISS;

- it synchronizes requests of the processor to the global timing model of SystemC, by issuing a SystemC wait-statement for the difference in time between ISS processing time and current SystemC time;
- it forwards load/store requests from the processor to the address decoder module;
- it determines how many clock cycles a memory access takes and returns this information to the processor;
- it returns the requested data (in case of a load) to the processor.

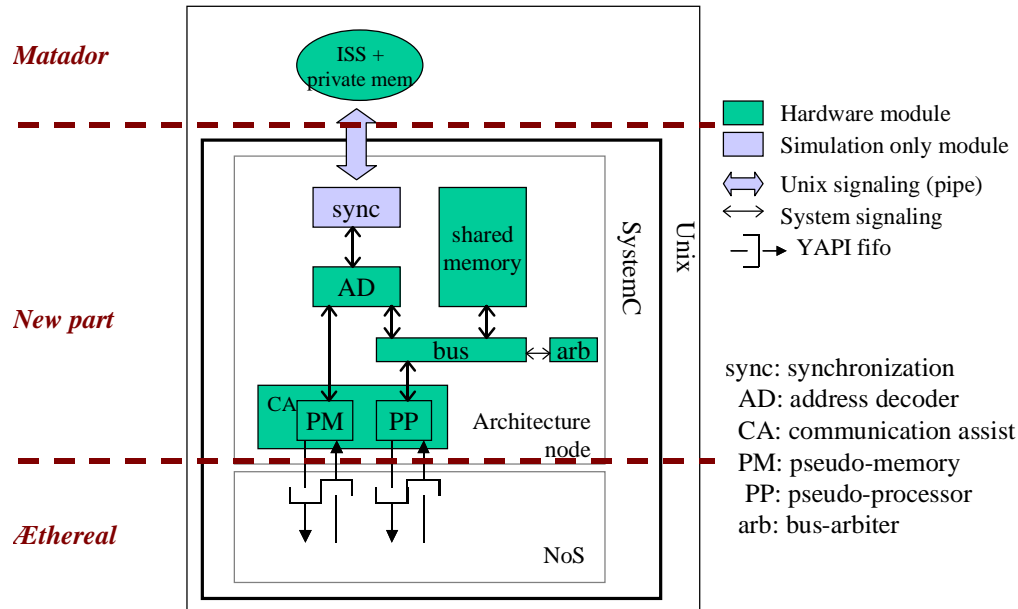


Figure 4-2: Global overview of developed simulator

Note, that since the NoS, implemented by means of the *Æthereal* simulation environment, as shown in Figure 4-2, also uses the SystemC global timing model, it can never happen that the NoS runs ahead of the processors. For example, it is not possible that a processor wants to perform a memory access at time x , while the NoS is already at time $x+y$, due to this global timing model of SystemC.

How does this synchronization to the global timing model of SystemC work? Suppose that an ISS wants to access a shared memory location. The synchronization module for this ISS does not immediately satisfy the request from the ISS, but posts this request for the future (using a SystemC wait-statement). Only when it is known for all ISSs when their requests should be processed, SystemC will pick up the earliest posted synchronization module, process it and sleep until the given ISS again says when it wants to do an access the next time. Therefore, for all ISSs, SystemC requests their following event, so that at any moment of time it is known to SystemC when they are due to be served, after which SystemC picks up the earliest ISS. If it is not known when an ISS wants to do an access, SystemC sleeps and does not do any processing, until this ISS replies. Furthermore, if an ISS is finished, it sends a “stop-message” to SystemC, to let SystemC know that it has finished.

A memory request is passed from the synchronization module to the address decoder module. This module decides, based on the global physical address (see Section 3.2), whether the access is to a local or to a remote shared memory location. If the address is local, then the shared memory is accessed via the local bus; otherwise, the request is passed on to the communication assist. The local bus is a simple single-transaction bus -only one master is allowed access the bus at any moment in time- with a bus-arbiter. This arbiter is necessary, since there are two masters connected to the bus: the address decoder and the pseudo-processor module. For arbitration, a round-robin scheduling algorithm is used. The round-robin algorithm is a fair scheduling algorithm; it guarantees that masters get the highest priority to access the bus in a cyclic order. A fair scheduling algorithm is chosen instead of for example a priority-based scheduling algorithm, since it is difficult to determine whether a local or a remote memory access should have priority.

The communication between the different SystemC modules in the architecture node is described by means of abstract interfaces using SystemC hierarchical channels ([SysC]). In this way, events or sequences of events on a group of wires are represented by a set of function calls in an abstract software interface, which enables higher simulation speed: it provides a higher level of simulation abstraction. The communication between pseudo-processor and pseudo-memory modules is implemented by means of YAPI FIFOs ([Koc02]), since the *Æthereal* environment only supports the mapping of FIFOs to a network path. For every network path that should exist between a pseudo-memory and a pseudo-processor, two FIFOs are necessary: one FIFO from the pseudo-memory to the pseudo-processor, and one from the pseudo-processor back to the pseudo-memory. The reason for this is the fact, that both pseudo-memory and pseudo-processor should be able to read packets from and write packets to the network.

4.3 Integration with *Æthereal*

For the network-on-silicon (NoS) part of the simulator, the *Æthereal* simulation environment is used, as shown in Figure 4-2. In this section we provide some important details of this simulation environment. Most of this section is based on [Pop02].

4.3.1 *Æthereal* Packet Format

The *Æthereal* environment currently only supports packet switching. Although longer packets are possible, the environment only uses packets that fit into one single *flit*: the smallest physical unit of information that can be transferred across a link. In Figure 4-3, the information encoded in a packet (or flit) is shown. It consists of four 32-bit words.

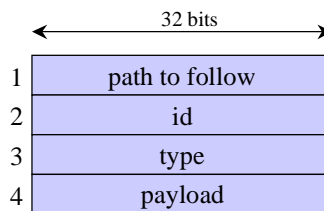


Figure 4-3: *Æthereal* packet format

The current router model of the *Æthereal* environment uses a simple routing algorithm, namely, source routing, which means that the encoded path is a vector. This vector contains for each router in the path the port number to which the packet should be routed. The first word of a packet consists of this path. The *id*-word identifies a YAPI FIFO channel, so that the receiving network interface knows to which FIFO channel the packet belongs. The *type*-word says what kind of payload the packet transports; e.g., type DATA means that it is (a piece of) a transaction data token. Finally, the *payload*-word contains the actual (piece of) transaction data token transferred across the network. Note, that in this way, every transaction data-token is split into packets containing a 4-byte payload. For example, if transaction data token T consists of 5 bytes, two packets will be sent.

4.3.2 Integration Assumptions

As already mentioned, the *Æthereal* simulation environment currently supports only the mapping of FIFO channels to a network path. Therefore, we specify the network connections in our simulation environment as small FIFO buffers in YAPI. In order to integrate our simulation environment with the *Æthereal* environment, we need to map the network connections onto pairs of network interfaces, as shown in Figure 4-4. In that example, the pseudo-memory unit of node 0 (PM0) communicates with the pseudo-processor units of nodes 1 and 2 (PP1 and PP2, respectively). The network connections beginning and ending at PM0 are mapped onto NI0, those beginning and ending at PP1 onto NI1 and the network connections beginning and ending at PP2 onto NI2. Note, that it is assumed that every pseudo-memory or pseudo-processor unit is connected to one network interface only, since it is possible to connect a module to multiple network interfaces. Furthermore, in this way, there is a

second way in which we use YAPI: to model network connections (we also use them at the application level for a completely other purpose, see Section 3.3). Note however, that in the future *Æthereal* transaction language (in stead of YAPI) will be used to describe these network connections.

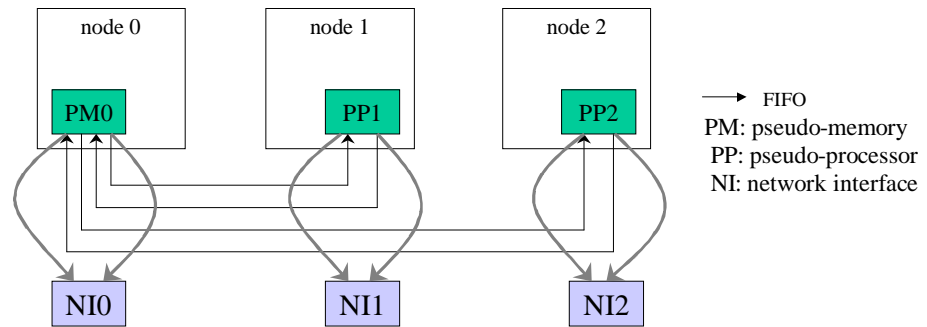


Figure 4-4: Integration with *Æthereal* environment

Figure 4-5 shows the global view of how a network connection mapped to the network is implemented in the *Æthereal* environment. The network is “introduced” to replace the “wire” that joined process PM with the bounded queue memory. The memory appears to be at the reading side. The communication actually goes in two directions: PM sends tokens through the network using the direct path from PM to PP. However, if PP is not fast enough to read the data, the queue memory will get full. In that case, PM stops sending tokens. Whenever PP wants to read K tokens from the output queue, it notifies PM by sending him a “credit” packet with the value K as a payload using the reverse path.

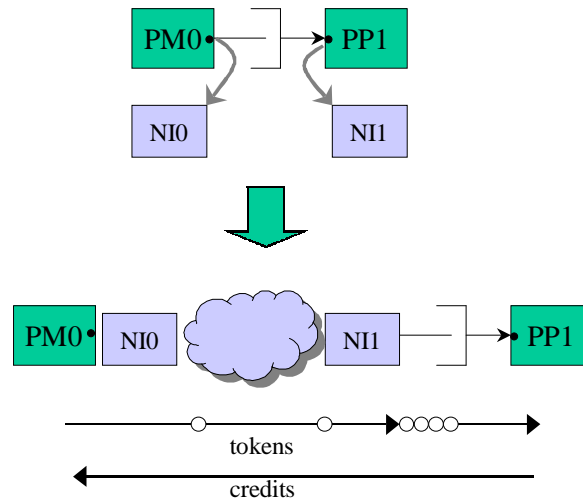


Figure 4-5: Mapping details

After the network connections have been mapped, the *Æthereal* environment will find the direct and reverse paths through the NoS for every network connection, using a path finding algorithm. The mapping of network connections has to be specified as a mapping specification in XML and serves as input for the *Æthereal* environment, together with a topology specification for the NoS in XML.

Note, that it is possible to run the simulator without integrating it with the *Æthereal* environment: just do not map the network connections onto network interfaces, but leave them as untimed FIFO channels.

Finally, in Appendix A, we present how a (posted) store operation “travels” through the implemented simulator.

4.4 Tuning the Architecture

Table 4-2 presents an overview of variable simulator parameters. These parameters allow us to “tune” the most important architecture parameters. Furthermore, they allow performing an architecture exploration for a certain application to determine the impact of architecture parameters on the performance of the application.

<i>Variable simulator parameters</i>	
Æthereal clock period	2 ns
SystemC default clock period	1 ns
maximum number of architecture nodes and ISSs	16
speedFactor	clock period of processor: $(0, \infty)$
clock period of bus and shared memory	clock period of bus and shared memory: $(0, \infty)$
shared memory size	total (physical) shared memory size (in bytes)
wait states of shared memory	number of clock periods an access to memory takes (default is 1)

Table 4-2: Simulator parameters

Currently, the simulator can only be used to measure performance in terms of clock cycles, since no power-/energy models are available for the Æthereal environment.

5 Case Study

In this chapter, we present an overview of the case study that we have performed. First, in Section 5.1, the goals of the case study are presented. In Section 5.2, we introduce the application that we used for the case study: JPEG decoding. Finally, in Section 5.3, the experiments and their results are presented.

5.1 Goals of the Case Study

The goals of the case study are defined as follows:

- test the functional correctness of the simulator;
- test the performance of the simulator with respect to
 - the number of processors used;
 - using the *Æ*thereal environment and the remote ISS approach;
- test the performance of the application with respect to architecture parameters (e.g., processor speed, network speed or (shared) memory size).

Furthermore, we want to show with this case study that our simulation environment is useful for measuring timing characteristics and for verifying analytically obtained timing characteristics. In that sense, the case study can be seen as a feasibility study.

Note, that for all experiments we only look at the performance in terms of clock cycles, since currently no power/energy-models are available.

5.2 JPEG Decoding

The JPEG decoder has been selected as a case study for three reasons. First of all, it is a sufficiently large application to be able to test the performance of the simulator. Second, this application is used as case study in other projects (e.g., [Koc02], [Stu02]), so it allows comparison of results. However, that was not the goal of this case study. Finally, one of the JPEG implementations used in [Koc02] was available for us, so we did not have to implement a JPEG decoder from scratch. In this section, we present details of the JPEG decoder used for our experiments.

A JPEG image consists of stripes, as shown in Figure 5-1. A *stripe* consists of MCUs. An MCU consists of (sub-sampled) chrominance (U, V) and luminance (Y) blocks, while a *block* consists of 8 by 8 pixels. Each color component is partitioned into these rectangular blocks of 8x8 pixels. A *minimum coding unit* (MCU) -the smallest group of interleaved data that completely describes a region of an image- usually consists of 4 Y, 1 U and 1 V blocks; however, exact ratios depend on the image coding.

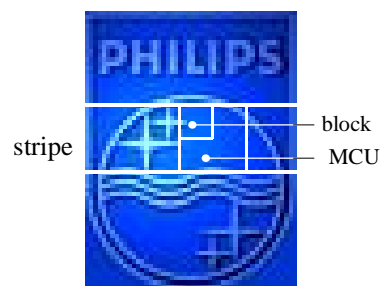


Figure 5-1: Elements of a JPEG image

Figure 5-2 shows the block diagram of a JPEG decoder. The input of the JPEG decoder is a byte stream connected to the DMX block. The DMX block de-multiplexes the byte stream into the tables required for the variable length decoding, tables required for the inverse quantization (IQ) and the bytes that must be parsed by the decoder. The variable length decoder (VLD) decodes the run length and Huffman encoded minimum coding units using the Huffman tables that were de-multiplexed by the DMX block. The VLD outputs decoded pixel blocks, which are dequantized by the inverse quantization (IQ) block. The IQ block uses for that the dequantization factors extracted from the input byte stream by the DMX block. The blocks then undergo inverse zigzag (IZZ) and two-dimensional inverse discrete cosine transformation (IDCT). The YUV2RGB block converts the blocks to stripes, applies vertical and horizontal scan rate conversion and color conversion from YUV to RGB.

For the reader interested in more details of JPEG, [Bha95] is recommended.

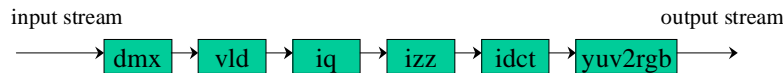


Figure 5-2: Block diagram of JPEG decoder

5.2.1 JPEG Process Network

In Figure 5-3, we present the JPEG process network that we used for our experiments (as described in [Kock02]) and its mapping onto the NUMA architecture. It consists of 5 processes: a frontend process, 2 IDCT processes, a raster and a backend process. The frontend process contains the DMX, VLD, IQ, and IZZ blocks shown in Figure 5-2. This frontend process reads a file in JPEG File Interchange Format (JFIF) from disk and writes pixel blocks consisting of 8x8 pixels to the IDCT processes. The two-dimensional IDCT of Figure 5-2 is split into 2 one-dimensional IDCT processes (on the rows and on the columns of a pixel block). The raster process reads the pixel blocks from the IDCT process, and contains the block-to-stripe conversion and the vertical and horizontal scan rate conversion of the YUV2RGB block (Figure 5-2), whereas the backend process contains the color conversion from YUV to RGB for every horizontal line of the output image. For this, the backend process reads a complete Y, U and V line from the raster process. Finally, the backend process writes the R, G and B streams to disk. Furthermore, the frontend process sends image information to both the raster and backend processes (e.g., horizontal and vertical size of the image, block ratios).

As presented in Section 3.3, every process is mapped onto one architecture node (shown in Figure 5-3 by the thick arrows) and every FIFO is mapped into the shared memory of the architecture node that has to read from this FIFO (not shown in the figure). For example, the FIFO from the frontend process to the idctRow process is mapped into the shared memory of node 1.

For the NoS topology, the simple routing network shown in Figure 5-3 is chosen. This network is chosen arbitrarily, in order to perform some first measurements. Obviously, the choice for a certain topology will have an impact on the measurements. However, the goal of this case study is not to find the most optimal mapping of a JPEG decoder on a NUMA architecture, but to get some insight in the performance of the simulator and the impact of the NoS on the performance of the application. Therefore, the choice of the NoS topology is not of real importance.

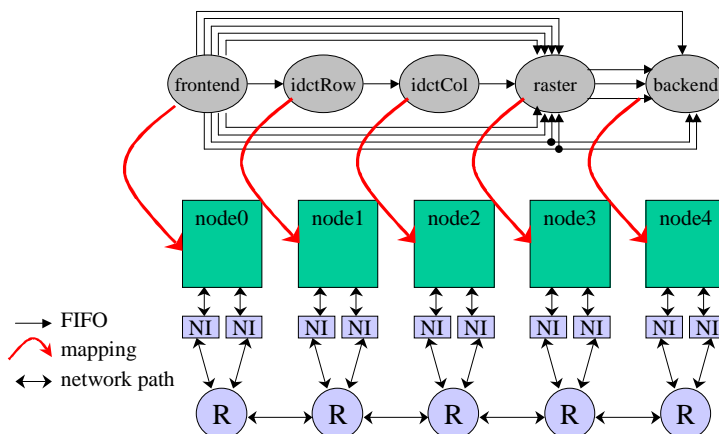


Figure 5-3: Mapping of JPEG process network onto NUMA architecture

5.3 Experiments

In the following, the experiments performed with the simulator are presented. For every experiment, the following parameters were used:

- the size of the shared memories for every architecture node is 16 kB;
- the size of the private data memories is 256 kB and the size of the private instruction memories 3 kB (we have measured these sizes during the experiments);
- the number of wait states for every memory is 1;
- the clock period of the NoS is fixed on 2 ns;
- the buffer size of the network interfaces is 8 transaction data tokens per network connection.

We used for all experiments the Philips.jpg image as shown in Figure 5-1: a JPEG image with a size of 50x67 pixels.

5.3.1 Experiment 1 - Impact of NoS

In the first experiment, we measure the number of ISS clock cycles needed for executing the JPEG decoder and the total simulation time, for different clock periods for the processor, bus and shared memory with respect to the clock period of the NoS. The clock periods for processor, bus and shared memory are kept equal, to simplify the experiment. Note, that in this case an increase of the clock period for the processor is the same as a decrease of the clock period for the NoS. The goal of this experiment is first of all to compare the execution time of JPEG on a multi-processor platform to the execution time on a single-processor platform. Secondly, the goal is to determine whether the NoS has an impact on this execution time.

We expect the execution of JPEG on a multi-processor platform to have a better performance in terms of clock cycles than the execution on a single-processor platform, since a multi-processor platform allows the exploitation of the parallelism present in the application. Furthermore, we expect the NoS to have a negative impact on the number of clock cycles per process: if the clock period of the NoS is slow compared to that of the processor, bus and memory, the NoS will slow down the total execution time due to a limited bandwidth and (a small) propagation delay.

The results of this experiment are presented in Figure 5-4; in Table 5-2 we present the total

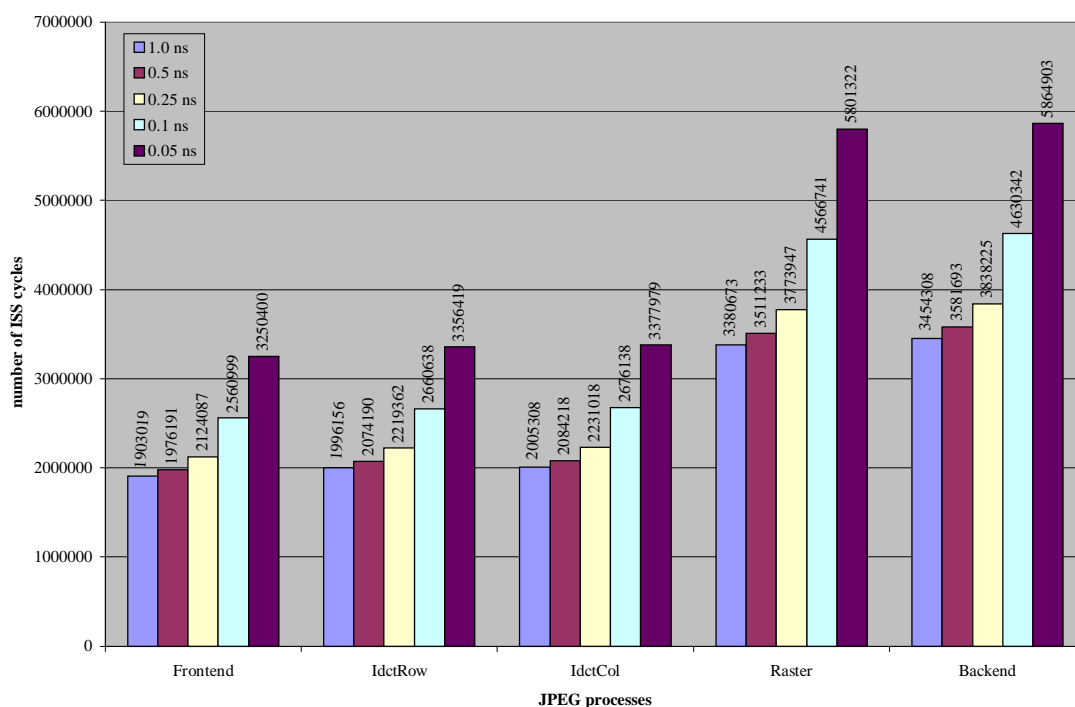


Figure 5-4: Results for experiment 1: total number of cycles per process with respect to ISS clock period

simulation time for each measurement. For comparison, we executed a 1-process JPEG application on a single ARM processor (ISS) with a clock period of 1 ns. The results of this execution are presented in Table 5-1.

<i>clock period (ns)</i>	<i>simulation time (s)</i>
1.0	406
0.5	319
0.25	243
0.1	155
0.05	199

<i>1-process JPEG</i>	
Number of ISS cycles	5730281
Simulation time	2 s

Table 5-1: Results for a 1-process JPEG application

Table 5-2: Simulation time for different clock periods

Comparing the results of Figure 5-4 for a clock period of 1 ns with the results of Table 5-1 shows that running this JPEG application on a multi-processor platform indeed results in a better execution time than running it on a single-processor platform. For this comparison, the maximum number of cycles in Figure 5-4 has to be compared to the number of cycles presented in Table 5-1, since the total execution time of the JPEG decoder is the number of cycles of processor with the longest execution time. However, for a clock period smaller than 0.05 ns, the performance of the multi-process JPEG becomes worse than that of the 1-process JPEG: in that case, the costs of data transfer outweigh the benefits of task parallelism. Furthermore, the NoS becomes a bottleneck if its clock period is more than 8 times slower than that of the processors. In the figure, this is shown by the non-linear growth of the total execution time for a clock period larger than 0.25 ns. Finally, Figure 5-4 shows that the raster and backend processes are the bottlenecks of this application. Reason for this is the fact that both processes have to perform conversions on three separate color streams. In order to increase the performance of this JPEG application, the parallelism in these processes could be made more explicit (e.g., separate the conversions into three parallel conversions, one for each color component).

As shown in Table 5-2, running the JPEG application on our simulation environment consumes much more *simulation time* than running it as one process on an ISS. Part of this is due to the integration with the \mathcal{A} ethereal environment, as shown in Table 5-3. In that table, we present the results of the execution of the JPEG application on our simulator, running it with and without the mapping of network connections onto the \mathcal{A} ethereal environment. In addition, the remote ISS solution for integration of the ISS into SystemC probably consumes a lot of simulation time (see Section 4.1), but we have not been able to produce numbers for that. However, more factors have an impact on the simulation time, shown by the decrease of the total simulation time in Table 5-1 for a decreasing clock period. This decrease is due to the fact that SystemC has less work to do: the number of \mathcal{A} ethereal cycles per given number of ARM cycles decreases by a factor of 2, 4, 10, 20, whereas the number of ARM cycles does not grow so fast in the beginning.

<i>Simulating with and without \mathcal{A}ethereal</i>		
<i>clock frequency (ns)</i>	<i>without \mathcal{A}ethereal</i>	<i>with \mathcal{A}ethereal</i>
1.0	137 s	421 s
0.1	99 s	164 s

Table 5-3: Simulation with and without \mathcal{A} ethereal environment (for Philips.jpg)

5.3.2 Experiment 2 - Communication vs. Computation

In the second experiment, we measure the number of ISS clock cycles for computation and communication per pixel block (8x8 pixels), for different clock periods of the processors, buses and shared memories with respect to the NoS. The goal of this experiment is to determine the impact of the NoS on the total number of cycles. Note, that a read operation⁴ is performed via the local bus, whereas a write operation is performed via the NoS (see Section 3.3). Therefore, the communication costs are expected to stay approximately the same for a read operation when changing the clock period of

⁴ Note, that read/write operations in this experiment are YAPI read/write operations, so they consist of several ISS load/store operations

processors, buses and shared memories. Furthermore, we have only measured the actual release and transfer costs for reading from or writing to a shared memory location, not the polling time spent in “acquiring” access to this shared location.

The results for this experiment are presented in Figure 5-5, for clock periods of 1.0 and 0.1 ns. Note, that for every process the average number of cycles is presented; however, the frontend process has a factor of 15 of variation in the total number of cycles, due to the VLD of the contents of the Y, U and V blocks.

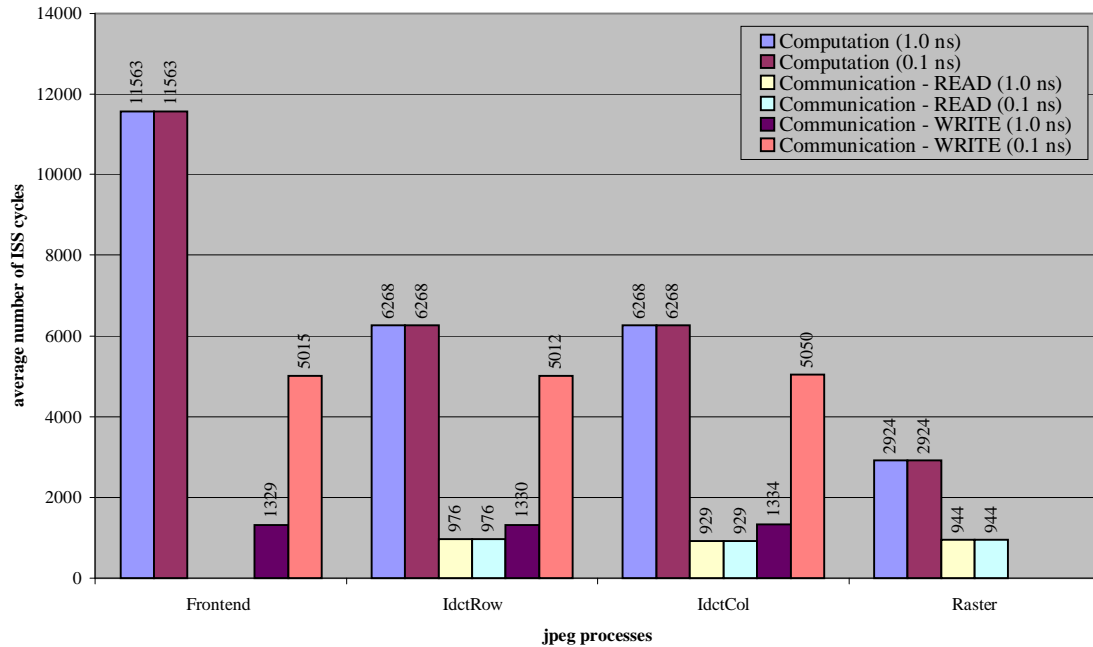


Figure 5-5: Results for experiment 2: computation vs. communication cost per pixel block (8x8 pixels)

As expected, the computation costs per process and the communication costs for a read operation (per process and also between processes) stay exactly the same, since the processor - bus/memory clock ratio stays the same. Furthermore, a decreasing clock period results in larger communication costs for a write operation. This increasing communication cost is for a large proportion due to contention in the network and splitting the pixel blocks into packets.

Another observation that can be made from Figure 5-5 is the fact that although the computation became a factor of 10 faster with respect to the communication, the communication (in case of a write operation) only became a factor 4 slower. The reason for this is the fact that for the communication the network was slowed down, but not the architecture node. Therefore, the total slow down for communication is less than a factor of 10.

5.3.3 Experiment 3 - Latency and Throughput

In this experiment, we measure the latency and throughput times for the processing of one MCU after deriving a theoretical value for both, for which we use the results of the previous experiments. The goal of this experiment is to test whether the simulator can be used to verify analytically obtained timing characteristics.

To process one MCU, the following nested loop could be executed in a process:

```

for i=1..nrScanComponents
  for j=1..h[i]
    for k=1..v[i]
      process 8x8 pixels
    endfor
  endfor
endfor

```

Figure 5-6 shows the task graph where this loop has been unfolded for 4 processes of JPEG assuming that `nrScanComponents` equals 3 (3 color components) and both `h` and `v` equal {2, 1, 1} (horizontal and vertical block size for every color component of an MCU; as is the case for Philips.jpg). This means that every MCU contains 6 pixels blocks in a 4:1:1 ratio: 4 Y, 1 U and 1 V block. The columns in Figure 5-6 correspond to the JPEG processes and the numbers assigned to the nodes are estimates of the computation times in kilocycles for a (processor) clock period of 0.1 ns, which are measured in the previous experiment (see results in Figure 5-5 for a clock period of 0.1 ns). The numbers are stable for all processes except for frontend, where the execution time can drastically change from MCU to MCU depending on the MCU contents (due to the VLD of the contents of the Y, U and V blocks). We report for frontend the average execution times for Y, U and V blocks separately (those numbers are not shown in Figure 5-5). The task graph executions are repeated, one after another, until a complete row of MCUs is processed (a stripe). When the stripe is ready, color conversion happens in the backend process (not shown in the figure, since we did not consider that here).

The longest path in the graph of Figure 5-6 determines the latency, the total time of execution for one MCU. Assuming that γ_{WR} is the time to push a pixel block through the network, γ_{RD} the time to read a pixel block and γ the sum of both, then the longest path, shown in the figure by the thick line, equals $(15 + \gamma_{WR}) * 4 + (7 + \gamma_{WR}) * 2 + (6 + \gamma) * 2 + \gamma_{RD} + 3$ kilocycles. For a clock period of 0.1 ns this latency equals 138 kilocycles, since γ_{WR} is 5 kilocycles and γ_{RD} 1 kilocycle, as shown in Figure 5-5 (communication READ and WRITE for 0.1 ns).

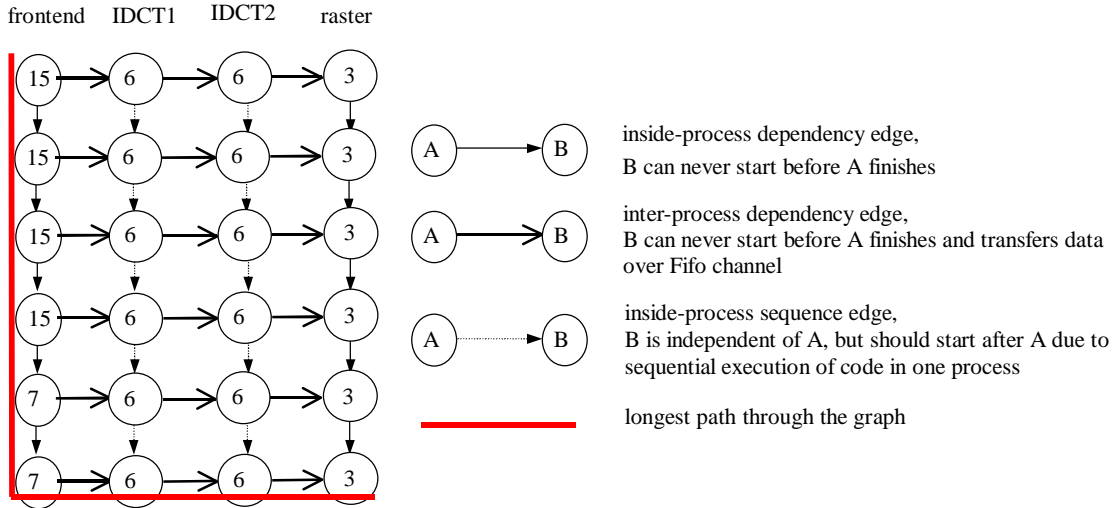


Figure 5-6: Theoretical latency for JPEG (clock period 0.1 ns)

The throughput is the number of MCU blocks processed per second by a processor, and equals $1/\text{data introduction interval}$. The process that has the slowest processing of one MCU block (and thus the maximum number of cycles) theoretically determines this data introduction interval. Looking at Figure 5-6, these data introduction intervals per process are as follows (total number of cycles per column):

$$\begin{aligned}
 \text{Frontend} &: 4(15 + \gamma_{wr}) + 2(7 + \gamma_{wr}) = 104 \text{ kilocycles} \\
 \text{idct} &: 6(6 + \gamma_{wr} + \gamma_{rd}) = 72 \text{ kilocycles} \\
 \text{Backend} &: 6(3 + \gamma_{rd}) = 24 \text{ kilocycles}
 \end{aligned}$$

Therefore, theoretically, the data introduction interval is 104 kilocycles, and thus the throughput $1/(104 \cdot 10^3 \cdot 0.1 \cdot 10^{-9}) = 96 \text{ kMCU} / \text{s}$.

The results of the measured latency and data introduction interval are presented in Table 5-4. As shown, these values are 10-15% off the theoretically determined values. However, since the execution times of the frontend can change dramatically from MCU to MCU and since we calculated the theoretical values for latency and throughput using average values for the execution times, we can state that these numbers more or less correspond to our predictions. Therefore, our simulation environment can be used to verify analytically obtained timing characteristics, as long as the execution

times are not too heavily data-dependent. Otherwise, the simulation environment can still be used to obtain timing characteristics of an application.

<i>Latency and throughput measurement</i>		
	<i>measured</i>	<i>predicted</i>
latency	155760 cycles	138000 cycles
throughput	111115 cycles	104000cycles

Table 5-4: Results for throughput and latency measurement (clock period 0.1 ns)

5.3.4 Experiment 4 - Three Parallel JPEGs

For this experiment, we ran three JPEG decoders in parallel on the NUMA architecture. In Figure 5-7, a “floor planning” view of the network topology and the placement of every architecture node are presented. The thick lines in this figure illustrate the mapping of the three JPEG decoders. The goal of this experiment is to determine whether the three JPEG decoders have influence on each other’s execution time, latency and throughput. For this, we measured the execution times in cycles, the latency and the throughput for three, two and one running (“awake”) JPEG decoders. Furthermore, we want to determine the impact of the total number of running processors on the total simulation time.

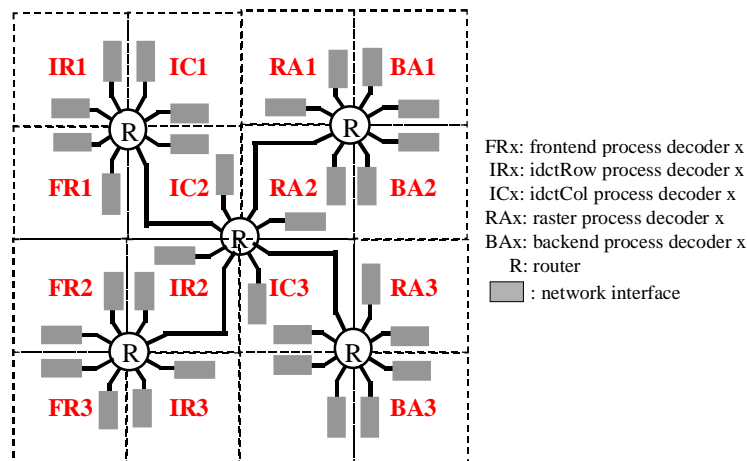


Figure 5-7: Three JPEG applications on NUMA architecture

In Figure 5-8, the results of the total execution times for JPEG decoder 3 for three, two and one running JPEG decoders are presented. These measurements are performed for a clock period of 0.1 ns for the processors, buses and shared memories. Table 5-5 shows the results of the latency and throughput measurements for decoder 3, whereas Table 5-6 presents the total simulation time for running the simulator with three, two and one decoders in parallel.

As shown in Figure 5-8, the total execution time for decoder 3 does not depend on the parallel execution of decoders 1 and 2: the total difference in execution time measured in clock cycles is less than 1%. In addition, according to Table 5-5, the latency and throughput do not depend on the parallel execution of the other decoders. However, we expected the execution time, latency and throughput to be influenced by this parallel execution, due to network congestion. Apparently, the bandwidth of the network is larger than the requested bandwidth of the JPEG application; therefore, there is no congestion. Shown in Table 5-6 is the total simulation time for running the simulator with 3, 2 and 1 JPEG decoders. From this table, we can conclude that the total number of processors running influences the total simulation time, which is not very unexpected. However, comparing these results with Table 5-1 for a clock period of 0.1 ns, we see a difference of a factor 5 in simulation time. This extra simulation time is for a large part due to the more complex NoS used for this experiment: compared to experiment 1, the total number of routers stayed the same, but the number of network interfaces and the number of network links changed.

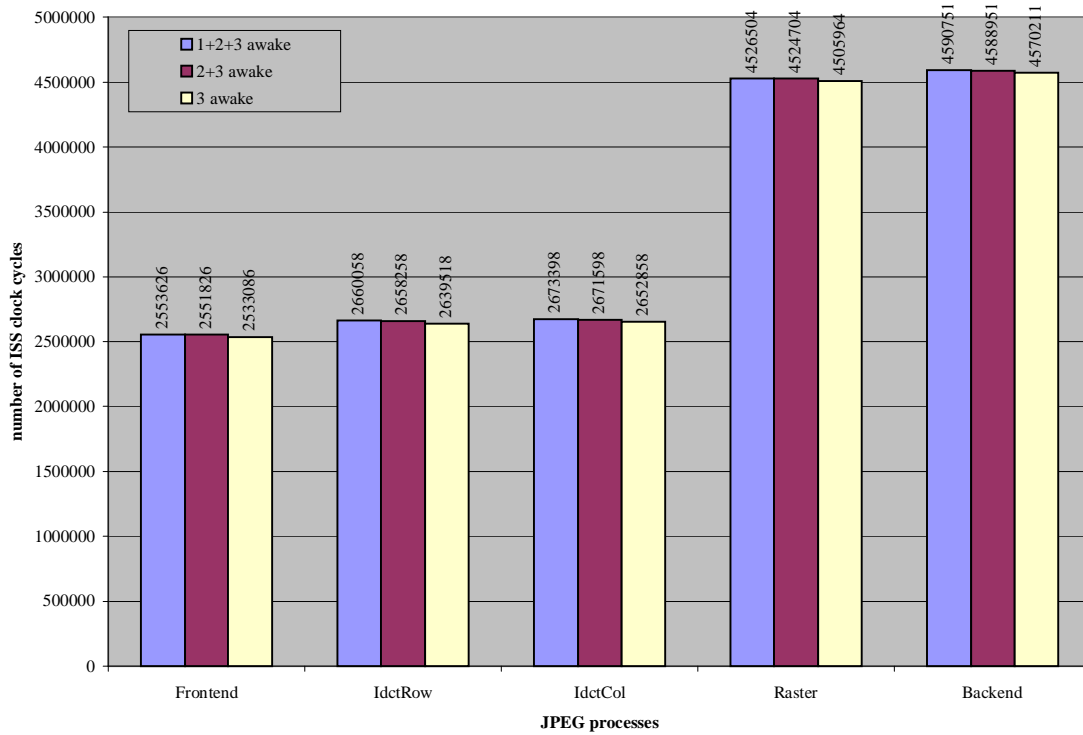


Figure 5-8: Results for experiment 4: number of clock cycles for JPEG decoder 3, with 3, 2 and 1 decoders running in parallel

<i>Latency and throughput measurements</i>		
	<i>latency (cycles)</i>	<i>throughput (cycles)</i>
1+2+3 awake	154425	109673
2+3 awake	154425	109673
3 awake	154425	109673

Table 5-5: Latency and throughput for JPEG decoder 3

<i>Total simulation time</i>	
1+2+3	1190 s
2+3	954 s
3	725 s

Table 5-6: Total simulation time for 3, 2 and 1 decoders running

6 Related Work

In this chapter, we present a small overview of other existing multi-processor simulation environments. However, it was not the purpose of this practical training project to perform a complete literature study. Therefore, we only mention a brief overview of the simulators and present a reference to more detailed information.

SPADE ([Lie01]) is a method and tool for architecture exploration of heterogeneous multi-processor systems. It employs a trace-driven simulation technique to co-simulate an application model with an architecture model in order to evaluate the performance of the combined system. This architecture model can be constructed from generic building blocks, such as processing resources, memory resources and communication resources. The processing resources in the architecture model take the traces generated by the application as an input.

The SPADE simulation environment simulates a distributed (private) memory architecture. Furthermore, it is a communication-centered simulator (as is our simulation environment), where communication parts are clearly separated from computation parts. Furthermore, the SPADE environment uses native processes as processor models and is therefore probably a faster simulation environment than to our simulator. In addition, the SPADE simulator allows dynamic scheduling of processes (not yet available in our simulator). A major disadvantage of the SPADE simulator, compared to our environment, is that it is limited to YAPI for application modeling. Furthermore, the processor models of the SPADE environment need timing parameters for every segment of code. Those have to be measured separately, using for example an ISS.

The *CAKE* project ([Str01]) provides a simulation environment for homogeneous tiles equipped with a large number of special purpose hardware functions that achieve high computational efficiency. These tiles communicate with each other via a communication network, whereas each of the tiles can be configured to execute a set of tasks. Assignment of tasks is statically determined, but within a tile tasks can dynamically arbitrate for resources such as memory and special purpose hardware. The programming paradigm used to model tasks is the process network programming paradigm.

The CAKE simulator uses ISS models (for the MIPS and Trimedia core) that are linked into the simulation backend, whereas in our simulation environment we use “remote” ISSs. The architecture simulated by the CAKE environment can be a mixture of both shared- and distributed memory architectures: the memories inside a tile are shared to all processors in that tile, but not to other tiles. For the communication between tiles message passing is used. Furthermore, the CAKE environment allows dynamic scheduling and task switches inside tiles. However, due to this dynamic scheduling and the complex architecture, it is very difficult to prove any performance characteristics of an application analytically, so the simulator is supposed to be run for long times to provide some confidence that timing constraints of an application will be met.

RSIM ([Hug02]) simulates shared-memory (multi-) processors built from processors that exploit instruction-level parallelism (ILP) and support dynamical scheduling. RSIM is execution-driven and models ILP-processors, a memory system, a multi-processor coherence protocol and interconnect, including contention at all resources. It is developed to study the influence of super-scalar processing nodes on the performance of shared memory multi-processor architectures. The processor architecture approximates the MIPS R10000. Possible configurations for this processor architecture range from single-instruction issue, in-order (static) instruction scheduling, blocking memory operations to multiple-instruction issue, out-of-order (dynamic) instruction scheduling and non-blocking memory operations. RSIM supports a two-level cache hierarchy with separate first-level data and instruction caches and a unified second-level cache. The main memory model allows interleaving and is accessed through a pipelined split-transaction bus. Furthermore, RSIM employs a full-mapped invalidation-

based directory cache-coherency protocol and it supports three memory consistency protocols: sequential consistency, processor consistency and release consistency. For remote communication, RSIM supports a wormhole-routed two-dimensional mesh-network. Finally, RSIM simulates applications compiled and linked for Sparc V9/Solaris using standard Sparc compilers and linkers at all optimization levels.

SimOS ([Ros97]) is an environment for studying the hardware and software of computer systems. It simulates the hardware of a computer system in enough detail to boot an operating system and run realistic workloads on top of it. To study long-running workloads, SimOS includes multiple interchangeable simulation models for each hardware component. By selecting the appropriate combination of simulation models, the user can control the tradeoff between simulation speed and simulation detail. This simulator is mainly used to characterize new architectural designs, to steer the development of operating systems and to evaluate the performance of applications.

The *Simics* simulation platform ([Mag02]) is based on the idea that reliable performance estimates require full system simulation. It attempts to strike a balance between accuracy and performance by modeling the complete application and providing a unified framework for hardware and software design. Therefore, it simulates a network of multiple, heterogeneous processors at the instruction-set level. In addition to processor models, Simics includes device models accurate enough to run with real firmware and device drivers.

The *Ptolemy* project ([Lee99]) studies heterogeneous modeling, simulation and design of concurrent, real-time systems. The focus is on embedded systems, particularly those that mix technologies, including for example analog and digital electronics, hardware and software and electronics and mechanical devices. The focus is also on systems that are complex in the sense that they mix widely different operations, such as signal processing, feedback control, sequential decision making and user interfaces. Ptolemy aims to combine different models of computation that govern the interaction between components in meaningful ways, so that complex systems can be designed in a coherent fashion.

7 Conclusions and Future Work

As presented in the previous chapters, a simulation environment for a multi-processor platform has been developed. In Chapter 2, we have presented a classification of multi-processor architectures, which resulted in a description of three architectures that are of interest for our work. Based on this classification and the requirements for the simulation environment (Section 3.1), the non-uniform memory access (NUMA) architecture is chosen as architecture for our simulator. A functional description of this architecture has been presented in Chapter 3. In Chapter 4, we have shown how we have integrated the ISS, processor and private memory models of the Matador environment into our simulation environment for the NUMA architecture. Furthermore, we have shown how we integrated the \mathcal{A} ethereal environment into our simulation environment. Finally, in Chapter 5, we have shown that we have built a simulator that can be used to obtain timing characteristics of an application. Furthermore, the simulator can be used to measure the performance of applications (currently only) in terms of execution time. This case study also reveals that running an application on a multi-processor architecture based on a NoS could be a feasible solution, however, a lot more work is required to really “reveal” it. Furthermore, simulation speed becomes a bottleneck, when simulating large applications running on our NUMA architecture simulator.

To conclude, we present a list of possible extension and enhancements for the simulator and the \mathcal{A} ethereal simulation environment, as well as a few possible future experiments.

Future work with respect to the implemented NUMA architecture simulator:

- continue working on the integration of the ARMulator ISS into SystemC, since this probably will increase the performance of the simulator;
- integrate other processor models into the simulation environment;
- provide means for modular plug-in of hardware modules into the simulation environment. In the current version, all hardware modules and their connections have to be specified by hand by editing the source code; this can for example be extended to a GUI that allows placement of hardware modules and their connections in a “plug-and-play” manner;
- extend the current simulator with power and energy models;
- allow the (dynamic) mapping of multiple tasks onto each processor. For this, a (static) scheduler or an operating system running on top of the simulation environment is necessary;
- currently, the timing model of the ISS and the simulation backend are not completely consistent. This should be fixed;
- step from YAPI to \mathcal{A} ethereal transaction language when the latter is provided by Philips; especially when the latter will support guaranteed throughput services and dynamic creation/deletion of connections;
- to get some data out of the simulation backend, the user has to add print statements in the source code of the hardware modules, i.e., modify the library. Instead, special modules or “monitors” should be created that could be seamlessly connected into different parts of the architecture and produce traces viewable by some graphical tool. An advanced option is to provide access to the state of these modules from the ISS command lines.

Future work with respect to the \mathcal{A} ethereal simulation environment:

- extend the simulation environment with power and energy models;
- provide a more generic network interface (\mathcal{A} ethereal transaction language), since the current interface only supports the mapping of YAPI FIFO channels onto a network path;

- modify the packet-format: in the current version only a payload of 4 bytes is allowed. In order to increase the performance, larger payloads should also be allowed;
- allow the “tuning” of parameters like for example the latency or clock period of the network.

Future experiments that could be performed with the simulator:

- experiment with a data-parallel JPEG application, to get more insight in the scalability in terms of parallelism on the NUMA architecture;
- perform the case study described in [Stu02] on the simulator and compare the results;
- find the bottleneck in terms of simulation time for the simulator;
- perform a (IPC conscious) task-scheduling experiment.

Bibliography

- [ARM] ARM, <http://www.arm.com> .
- [Ben02] L. Benini et al., “Legacy SystemC Co-Simulation of Multi-Processor Systems-on-Chip”, *Proc. ICCD 2002 – Computer Design: VLSI in Computers & Processors Conference*, pp. 494-499, Sep. 2002.
- [Bha95] V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards*, Kluwer Academic Publishers, 1995.
- [Cha99] H. Chang et al., *Surviving the SOC revolution – A guide to platform-based design*, Kluwer Academic Publishers, Boston, Ma, 1999.
- [Cul99] D.E. Culler, J.P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufman Publishers, San Francisco, CA, 1999.
- [Hug02] C.J. Hughes et al., “RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors”, *IEEE Computer*, 35(2): 40-59, Feb. 2002.
- [Kah74] G. Kahn, “The Semantics of a Simple Language for Parallel Programming”, *Proc. of the IFIP Congress 74*, 1974.
- [Koc02] E. de Kock, G. Essing, “Y-Chart Application Programmer’s Interface: Application Programmer’s Guide”, version 1.1, *Philips internal*.
- [Lee99] E.A. Lee et al., “Overview of the Ptolemy project”, *ERL Technical Report UCB/ERL M99/37*, University of California, Berkeley, July 1999.
- [Lie01] P. Lieverse et al., “System Level Design with SPADE: an M-JPEG Case Study”, *Proc. ICCAD 2001*, Nov. 2001, San Jose, CA.
- [Mag02] P.S. Magnusson et al., “Simics: A Full System Simulation Platform”, *IEEE Computer*, 35(2): 50-58, Feb. 2002.
- [Man00] H. De Man, “System Design Challenges in the Post-PC Era”, *Keynote address, 37th ACM/IEEE Design Automation Conf.*, Los Angeles, June 2000.

- [Mar02] P. Marchal et al., “Matador: an Exploration Environment for System Design”, *Journal of Circuits, Systems and Computers*, 11(5): 503-535, Oct. 2002.
- [Pop02] P. Poplavko, “Yapi/Æthereal Co-Simulation Environment”, draft, *Philips internal*.
- [Rij01] E. Rijpkema, K. Goossens and P. Wielage, “A Router Architecture for Networks on Silicon”, *Proc. Progress 2001, Second Workshop on Embedded Systems*, Nov. 2001. <http://www.dcs.ed.ac.uk/home/kgg> .
- [Ros97] M. Rosenblum et al., “Using the SimOS Machine Simulator to Study Complex Computer Systems”, *ACM Trans. on Modeling and Computer Simulation*, 7(1): 78-103, Jan. 1997.
- [Str01] P. Stravers, J. Hoogerbrugge, “Homogeneous Multiprocessing and the Future of Silicon Design Paradigms”, *Proc. Int. Symp. on VLSI Technology, Systems and Applications 2001*, pp. 184-187, April 2001, Hsinchu, Japan.
- [Stu02] S. Stuijk, “Concurrency in Computational Networks”, Master’s Thesis, *Faculty of Electrical Engineering, Eindhoven University of Technology*, The Netherlands, 2002. <http://www.ics.ele.tue.nl/~sander> .
- [SysC] SystemC Functional Specification. <http://www.systemc.org> .
- [TRI] Trimaran, <http://www.trimaran.org> .
- [Yan01] P. Yang et al., “Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs”, *IEEE Design and Test Special Issue on Application-Specific SOC Multiprocessors*, 18(5): 46-58, Sep. 2001.

List of Figures

Figure 1-1: Example of multi-processor platform.....	1
Figure 2-1: Possible locations of cache	6
Figure 2-2: General overview of the distributed memory platform	6
Figure 2-3: Typical configuration of a shared memory hierarchy	8
Figure 2-4: Sharing of data in a multi-processor environment	9
Figure 2-5: Cache coherency protocols	9
Figure 2-6: Example of consistency problems in a multi-processor environment	10
Figure 3-1: NUMA architecture structure	14
Figure 3-2: Memory model for NUMA architecture.....	14
Figure 3-3: Issuing a load to a remote memory location	15
Figure 3-4: Example of network topology.....	16
Figure 3-5: Example of process network with a 1-entry FIFO buffer	17
Figure 3-6: Mapping of process network on architecture.....	17
Figure 4-1: General simulation model.....	19
Figure 4-2: Global overview of developed simulator.....	21
Figure 4-3: Æthereal packet format.....	22
Figure 4-4: Integration with Æthereal environment	23
Figure 4-5: Mapping details.....	23
Figure 5-1: Elements of a JPEG image	25
Figure 5-2: Block diagram of JPEG decoder	26
Figure 5-3: Mapping of JPEG process network onto NUMA architecture.....	26
Figure 5-4: Results for experiment 1.....	27
Figure 5-5: Results for experiment 2.....	29
Figure 5-6: Theoretical latency for JPEG.....	30
Figure 5-7: Three JPEG applications on NUMA architecture	31
Figure 5-8: Results for experiment 4.....	32

List of Tables

Table 4-1: Simulation granularity levels	19
Table 4-2: Simulator parameters.....	24
Table 5-1: Results for a 1-process JPEG application	28
Table 5-2: Simulation time for different clock periods	28
Table 5-3: Simulation with and without Æthereal environment	28
Table 5-4: Results for throughput and latency measurement.....	31
Table 5-5: Latency and throughput for JPEG decoder 3	32
Table 5-6: Total simulation time for 3, 2 and 1 decoders running	32

Appendix A

In this appendix we show how a (posted) store operation to a remote memory location “travels” through the simulator.

