Mapping Streambased Applications to an Intel IXP Network Processor using Compaan

—MASTER'S THESIS—
of

Johan Walters
student nr.: 9917047
jwalters@liacs.nl

and

David Snuijf
student nr.: 9921354
dsnuijf@liacs.nl

Supervisors:
Dr. ir. Bart Kienhuis
Dr. ir. Todor Stefanov

LIACS
Leiden University

August 31st, 2006

Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**
- 1.1 Problem Description . . . 3
- 1.2 Solution Approach . . . 6
- 1.3 Related Work . . . 8
  - 1.3.1 Tools Using COMPAAN . . . 8
  - 1.3.2 Alternative Programming Models . . . 9
- 1.4 Thesis Organization . . . 9
- 1.5 Workload Partitioning . . . 10

2 Application Modeling **11**
- 2.1 Static Affine Nested Loop Programs . . . 11
- 2.2 Kahn Process Networks . . . 12
- 2.3 The COMPAAN Compiler . . . 13
- 2.4 Example Transformation . . . 14

3 The Intel IXP Network Processor **19**
- 3.1 Choosing the Intel IXP Network Processor . . . 19
- 3.2 The Intel IXP Network Processor Family . . . 20
- 3.3 Architecture Overview of the IXP2400 . . . 21
  - 3.3.1 Programmable Processing Units . . . 21
  - 3.3.2 Threads and Thread Arbitration . . . 24
  - 3.3.3 Registers . . . 26
  - 3.3.4 Memory . . . 26
- 3.4 Media and Switch Fabric Interface . . . 28
- 3.5 Signals . . . 28
- 3.6 Programming the IXP . . . 30
  - 3.6.1 Programming Languages and Environment . . . 30

4 Mapping **33**
4.1 Process Mapping . . . 33
4.1.1 Processes to Processors . . . 33
4.1.2 Cardinality . . . 34
4.1.3 Latencies . . . 35
4.1.4 Processes to Threads . . . 36
4.1.5 Resources . . . 37
4.1.6 Strategies . . . 37
4.2 Channel Mapping . . . 38
4.2.1 FIFO Characteristics . . . 39
4.2.2 Hardware Options . . . 40
4.2.3 Strategies . . . 43
4.3 Implementation . . . 44

5 Code Generation **49**
5.1 FIFO code . . . 49
5.2 Process code . . . 50
5.2.1 Microengine Code Example . . . 51
5.3 Implementation . . . 52

6 Receive and Transmit **55**
6.1 Receiving packets . . . 55
6.2 Reading and Assembling Packets . . . 58
6.2.1 Reading Packets . . . 58
6.2.2 Creating Packets . . . 59
6.3 Transmitting Packets . . . 60
6.4 Speed . . . 60
6.5 Stop Signals . . . 61

7 Experiments and Results **65**
7.1 Method . . . 65
7.2 QRvr . . . 66
7.3 FDWT . . . 68

8 Improvement Suggestions **71**

9 Conclusions **75**

Appendix **76**

A Example Microengine C File **77**

Bibliography **81**

List of Figures

1.1 Market requirements 2
1.2 Productivity gap 3
1.3 Overview 4
1.4 The mapping challenge. 5
1.5 Design flow 6

2.1 Example code describing Static Affine Nested Loops 11
2.2 Simple Kahn Process Network 13
2.3 Matlab QRvr code 16
2.4 The KPN of QRvr 17

3.1 IXP2400 22
3.2 IXP2400 Microengine Block Diagram 23
3.3 Threads 24
3.4 Cooperative Threads Pseudocode 25
3.5 Signal to wait for memory operation 29
3.6 Synchronisation between microengines using signals 29

4.1 Schema 45
4.2 Schema with mapping 46
4.3 FIFO options 47
4.4 Platform Class Structure 47

5.1 Syntax Tree 51
5.2 Process C Code 53
5.3 UML of Visitor 54

6.1 Flow 56
6.2 Flowchart of a packet reassembly from mpackets 57
6.3 Receiving on one sourcenode 59
6.4 Sending one mpacket 60
6.5 Flowchart assembling an mpacket into a packet to be transmitted 63

7.1 Conformation of processes to bottleneck speed in QRvr 67

8.1 Receiving on two sourcenodes 73

List of Tables

3.1 The Overview of the components of the IXP2325, IXP2400, and IXP2855 Processors 21
3.2 Properties of the four IXP2400 Processor Memories 27

4.1 Mapping of nodes from Figure 4.1 onto IXP2400 39

7.1 Performance results QRvr, 1 run 66
7.2 Speed results QRvr 68
7.3 Results FDWT 68

Chapter 1

Introduction

Nowadays, single processor embedded system architectures fail to reach modern performance requirements. It no longer suffices to simply increase the amount of gigahertzes to improve performance efficiency. Multiprocessor Systems are therefore increasingly becoming important, as it is much easier to increase the quantity of processors than the quality, in terms of performance. In contrast to singleprocessor architectures, multiprocessor systems allow to exploit parallelism between tasks and data flows. Any two operations of an application running in parallel saves execution time, and thus increases performance efficiency.

Figure 1.1 illustrates the shortcoming of single processors when comparing performance to modern market requirements. The amount of 8-bit Multiply-Accumulate operations per second (MAC/s) required by modern applications such as Video over IP and wireless telephony systems increases at a higher rate than the performance of general purpose singleprocessor systems. This causes an increasing gap, which can only be closed by using multiprocessor systems.

Multiprocessor hardware is becoming increasingly complex to allow for even greater performance. However, programming efforts fall behind to exploit this increased complexity. This is called the *productivity gap*, as shown in Figure 1.2. One of the reasons that programming multiprocessor systems is much more troublesome than programming single processor systems, is that designing the parallel application specification is extremely hard. First of all, parallel programming languages hardly exist or fall short. Secondly, a good and easy to use programming language will probably never exist, as the human mind is hardly capable of easily conceiving parallel fine-grained tasks to efficiently operate together. Only well-trained experts might be able to do so, yet the amount of required programming efforts worldwide vastly supersizes the capacity of all available experts.

A more realistic approach is to use a translator to convert a sequentially specified application to a parallel specification. Such a translator would automatically reveal the parallelism within the data flow of the sequentially specified application. Luckily, such

a translator exists in the form of the COMPAAN [1] compiler. It transforms sequentially specified Matlab [2] code into an equivalent parallel specification.

Still needed is a way to map the parallel specification onto the platform of choice. Various back-ends have been developed for the COMPAAN compiler to target different platforms. Both LAURA [3] and ESPAM [4] convert the parallel specification produced by COMPAAN to an FPGA[1] platform.

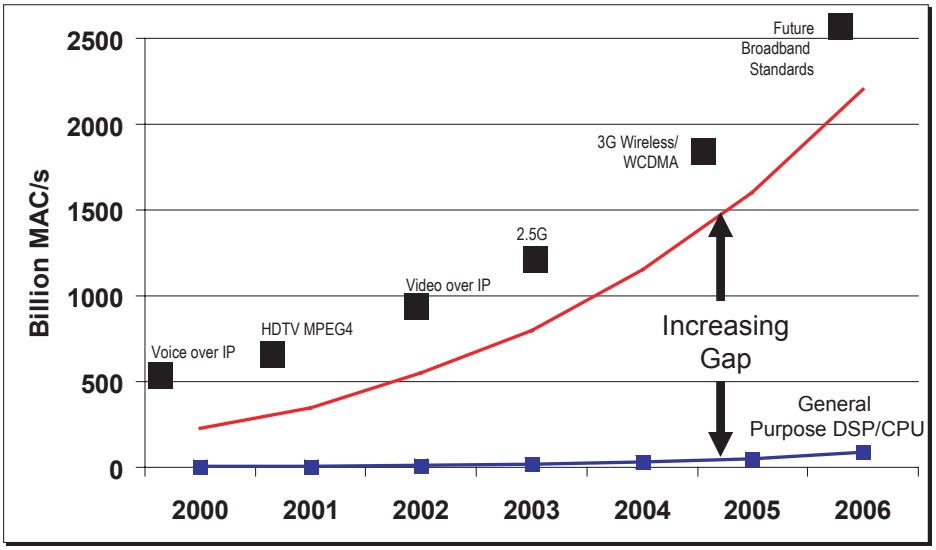

Figure 1.1: Market requirements
The trend is that there is a ferocious appetite for more computing power.
Source: TI, Xilinx 1 MAC = 8 bit Multiply-Accumulate

Of course, many more multiprocessor systems exist. In the field of network data processing, such systems are being developed to cope with high network speeds. For example, the Intel IXP network processor product line [5] was designed for this purpose, being programmable to do whatever task is needed in the network. It has a higher clock frequency than for example an FPGA (currently up to 1.4 GHz instead of about 150 MHz for a MicroBlaze processor on the Xilinx [6] Virtex Pro II FPGA). Another advantage is that computer network hardware is much more common than hardware traditionally used for signal processing applications, leading to favourable pricing of the hardware. In the case of the Intel IXP network processor series, a relatively straight-forward mapping from a parallel application specification to the platform seems possible, making it interesting to explore this possibility.

[1] Field Programmable Gate Array

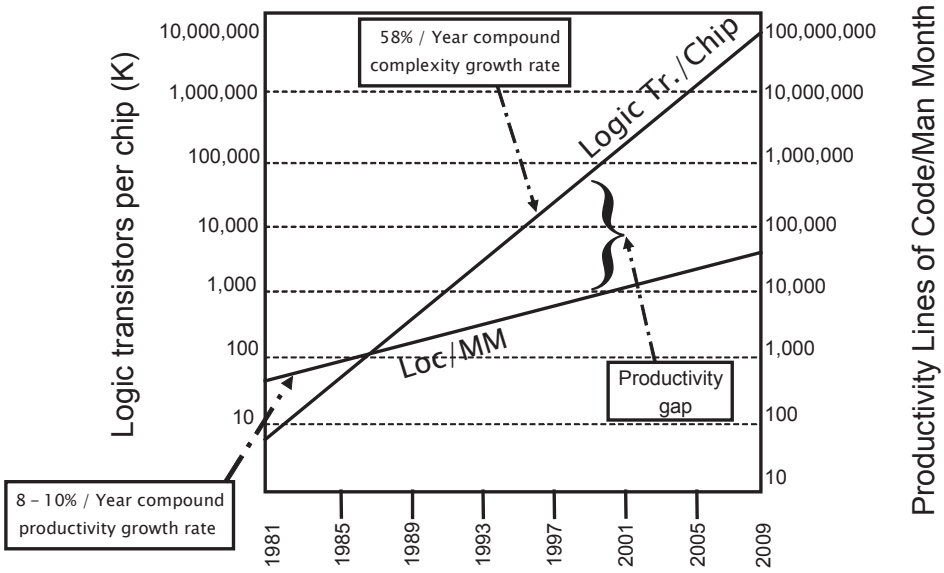

Figure 1.2: Productivity gap
Efficient tooling is required to close the productivity gap.

1.1 Problem Description

Streaming applications like DSP [2], software-defined radio, aerospace and defence systems, medical imaging, computer vision, speech recognition, cryptography, and bioinformatics, usually consist of a variety of complex algorithms. They perform highly repetitive arithmetic tasks and require high throughput rates. Examples are FFT [3], DCT [4], and audio/video decoders. Multiprocessor platforms are well-suited for this domain of computing, as they provide the required performance. Also, being programmable or (re)configurable, the application designs are more maintainable than dedicated designed hardware.

When using 'off the shelf' hardware to save costs, choice is often made for hardware specifically designed for this realm of computing, like an FPGA. However, the world of network data processing has come up with their own multiprocessor hardware suiting their own needs, but not often being used in the domain sketched above. For example, the Intel IXP Network Processor product line offers high-speed and high-throughput multiprocessor processing of network data. It is designed to process streaming data, which is also a typical aspect of the signal processing application domain. This feature forms an advantage in comparison with FPGA hardware, where all the streaming processing structures still have to be implemented in relatively slow programmable logic.

[2] Digital Signal Processing
[3] Fast Fourier Transform
[4] Discrete Cosine Transform

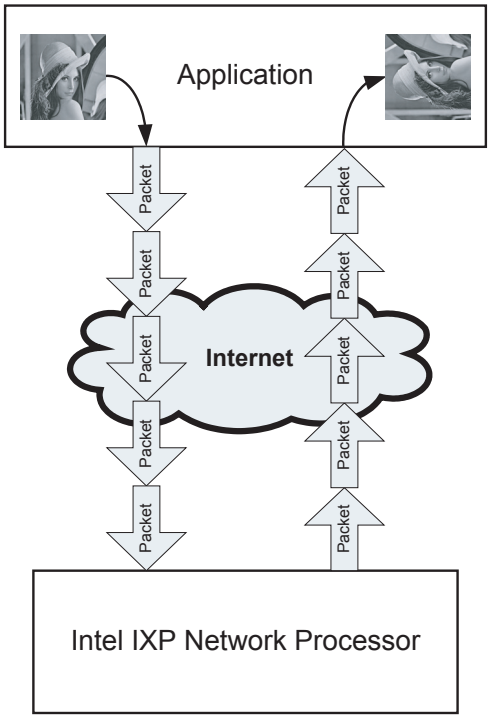

Figure 1.3: Overview

This is exactly where the challenge lies: effectively using an Intel IXP Network Processor for the domain of signal processing applications. An example of how an Intel IXP Network Processor can be used for streaming applications, is illustrated in Figure 1.3. An application sends data over a network connection to the network processor. There, it is processed and sent back to the application over the network.

The real problem is that Intels IXP Network Processors are hard to program, much like other complex multiprocessor systems. Very specific and detailed knowledge of the hardware is needed to be able to program the device, such that only platform experts will be able to do so. Intel does take measures to ease IXP programming, like offering the IXA [5] Portability framework[7]. In this way, functionality can be constructed more modular. This functionality is however mostly targeted at network-specific tasks. As we experienced, programming the IXP is still hard, slow and error-prone, even when not taking into account the difficulty of modeling concurrency in an application. Many assumptions from the world of general-purpose programming do not apply; for example, there is no unified memory model or automatic caching of data and alignment has to be constantly checked by the programmer. Therefore, a higher level of programming is required.

[5] Internet Exchange Architecture

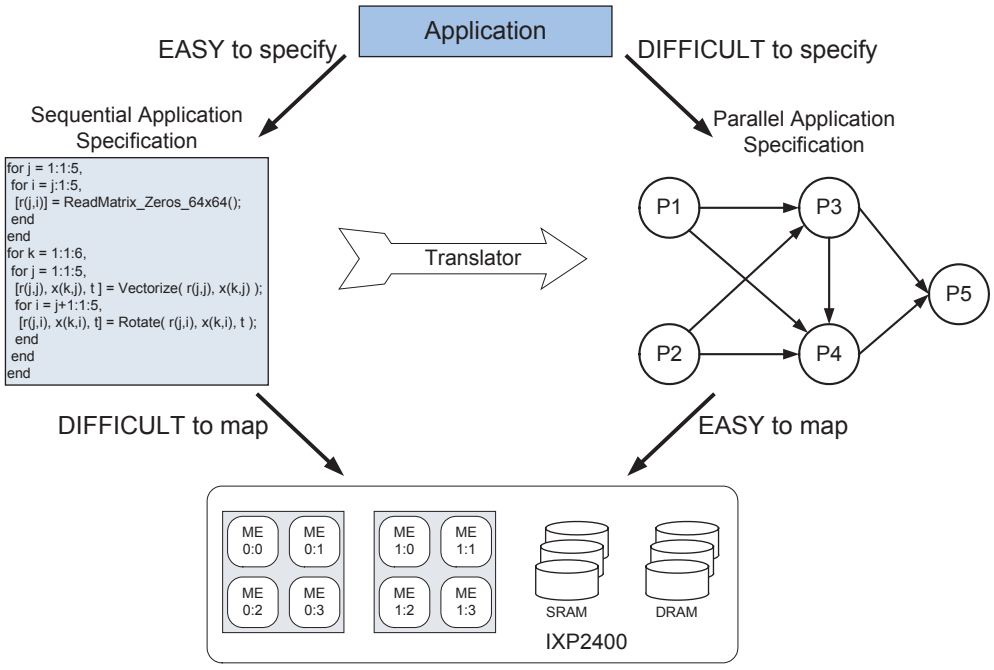

Figure 1.4: The mapping challenge.

Both for easing the modeling of concurrency in an application, as for mapping the concurrent model to an Intel IXP Network Processor platform, methodologies are developed. These methodologies together provide a high level of programming the Intel IXP Network Processor. The challenge of mapping an application specification to a multiprocessor platform is shown in Figure 1.4. The specification of an application is easy to specify and to reason about, because of the single line of control. It can be expressed with a sequential programming language like Matlab. Such sequential programming languages do not reveal the parallelism in the application due to their sequential nature. Therefore, mapping this specification directly onto multiprocessor hardware like the IXP2400 is useless, as only a fragment of the hardware will be effectively used.

Mapping an application specification with concurrent processes onto the hardware is more useful, as each concurrent process can be mapped onto a programmable processing unit of the hardware. However, it is difficult to specify such a parallel specification ‘by hand’.

It is more effective to combine the easy application specification using sequential programming languages, and the straightforward and efficient mapping of concurrent processes onto multi-processor hardware. To do so, sequential application specifications need to be translated into equivalent parallel specifications. For example, the Compaan compiler will reveal the concurrency of the data flows in the sequential specification and generate a network of concurrent processes.

The last step is the mapping and implementation of the concurrent processes onto the platform of the IXP2400 Network Processor. The goal for this thesis is to investigate the platform hardware and the programming environment, and to use this knowledge for producing and implementing a mapping strategies of parallel applications generated by COMPAAN. Hereby we investigate whether the mapping can be performed automatically onto the IXP. Furthermore, we measure the performance of a mapped result.

1.2 Solution Approach

In this section, we provide an overview of our solution approach with the techniques we have developed to map a parallel application specification onto the IXP2400 hardware.

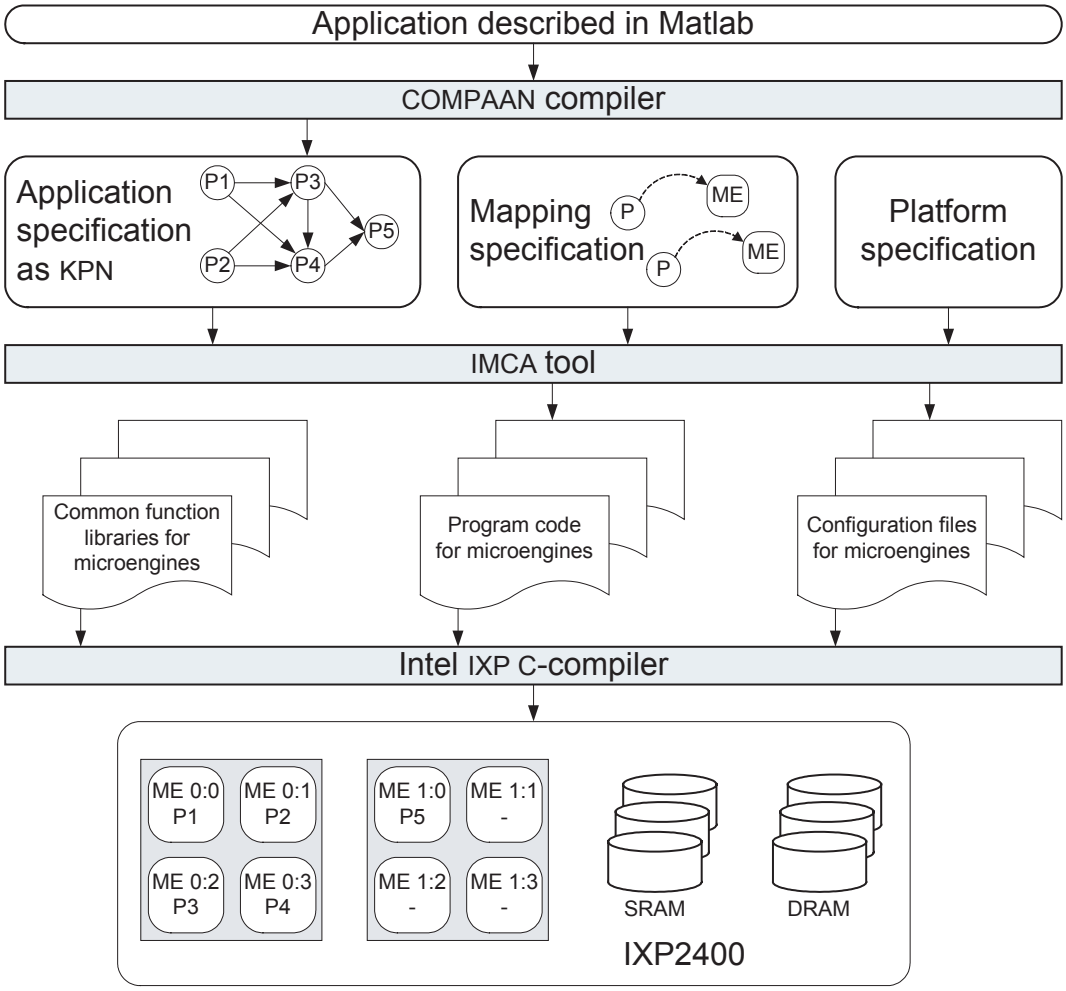

Figure 1.5: Design flow

A general overview of the mapping process is given in Figure 1.5. The design flow consists of three elements: COMPAAN, IMCA (*IXP Mapper for* COMPAAN *Applications*) and the Intel C Compiler for Intel Network Processors. Together, they form a fully automated process for converting sequential Matlab code into an implementation that runs on the IXP2400 Network Processor platform. Our contribution is the IMCA tool, which links the COMPAAN data flow to that of the Intel compiler.

In the first step, the COMPAAN compiler translates a sequential application specification in Matlab to an equivalent specification in the form of a Kahn Process Network (KPN). This is the main input for the IMCA tool. Other input needed are the platform specification and the mapping specification.

- The *application specification* describes how concurrent processes operate together using FIFO channels for communication. The application is specified as a KPN, in the format as is generated by COMPAAN. More information about the KPN model of computation is given in chapter 2.

- The *platform specification* describes the hardware components of the IXP Network Processor platform. The most important variable is the number of microengines. Other variables, such as the number of hardware-assisted FIFO queues and the size of the scratchpad memory, are in practice the same for all current models of Intel's IXP Network Processors. Nevertheless, they are configurable too. A detailed overview of the hardware is provided in chapter 3.

- The *mapping specification* describes how the elements of the KPN specification should be mapped on the hardware of an IXP Network Processor. Several mapping strategies are possible. For example, the processes of the KPN can be mapped onto the threads of the microengines in some round-robin order, or special considerations such as the next-neighbour order of the engines can be taken into account to provide extra FIFO implementation options. Chapter 4 deals with the subject of mapping strategies.

The IMCA tool will map the application specification onto the specified platform, using the strategies provided in the mapping specification. This mapping is used to generate code and other files implementing the application on the hardware. More on the subject of the code generation process is described in chapter 5. The files that are generated by IMCA are as follows:

- *Common function libraries for microengines* are not generated based on input of the IMCA tool, but are essential to implement common functionality needed by all microengines. For example, implementations of the various FIFO types on the IXP2400 hardware, as well as a general interface for accessing these FIFO types are included. The code is written in the microengine C language.

- *Program code for microengines* consists of microengine C files, one file per microengine on which processes from the KPN have been mapped. This code implements the selection and usage of all incoming and outgoing FIFO channels as well as the assign statements and function calls of the mapped processes. This code makes use of the common function libraries.

- *Configuration files for microengines* provide additional data for each microengine, such as which microengine C files are mapped onto which microengine and the number of threads that should be active. The extension of these files is `.list`.

These input files will be compiled and linked into a single executable `.uof` file using the Intel C Compiler for Intel Network Processors. This file can be used with either the platform simulator provided by Intel or on the hardware itself.

1.3 Related Work

There are two types of related work to our research. First, there are ESPAM and LAURA, which also serve as back-ends to the COMPAAN compiler, but are targeted at different platforms. Second, there are other programming methodologies for Intel IXP Network Processor platforms.

1.3.1 Tools Using COMPAAN

The most closely related work to ours is that of ESPAM [4]. Like IMCA, ESPAM uses COMPAAN and converts KPN specifications into a multiprocessor implementation. The output of ESPAM is a so called RTL-level [6] implementation, which can be converted to an FPGA implementation by a commercial synthesizer tool. The RTL-level specification is in essence a synthesized heterogeneous multiprocessor platform specification. This is in contrast to our work, as our mapping is targeted at a fixed homogenous multiprocessor platform specification.

The COMPAAN/LAURA design flow is also similar to that of ESPAM and that of our work as described Figure 1.5 on page 6. It uses COMPAAN to generate a KPN specification, which is converted to an FPGA implementation. The reported results are only for processor-coprocessor architectures and not for multi-processor architectures.

The benefit of our work with relation to LAURA and ESPAM is that FPGA platforms typically operate at much lower clock frequencies than the IXP2400 or the IXP2800.

[6] Register Transfer Language

1.3.2 Alternative Programming Models

Most programming efforts for the Intel IXP Network Processors is done using the microengine assembly programming language. It allows for very efficient use of the hardware and offers an enormous flexibility to the programmer. However, programming using the microengine assembly language is very complex, time-consuming and error-prone. Designing applications generally takes months. This is partially due to the fact that the programmer has to design all concurrency manually. Using the Intel C compiler for Intel Network Processors offers a higher level of programming, but still most of the hardware specific knowledge is required. An adaptation of the C language is used, called *microengine C*. An example of an application implemented using microengine assembly and microengine C is the FFPF/Streamline application [8].

To improve programming efforts for the Intel IXP Network Processors, other programming models have been developed. One of such programming models is NP-Click [9]. It offers a higher level of abstraction of the underlying hardware of the IXP1200 network processor. It is mainly targeted at network processing applications, offering support for header verification, route table lookup, and so on. Programming is mainly done by 'clicking' data flow streams together.

Another effort for improving programming simplicity on Intel network processors is the μL programming language and the μC compiler by Network Speed Technologies [10][11]. Like NP-Click, it offers a high level of abstraction to the hardware, where the programmer focuses on the data streams of network processing applications.

One aspect in common between NP-Click, μL/μC and our work is that all three seek for effective tooling for closing the implementation gap by providing a programming model with a high abstraction level. All three tools are suitable for defining streaming data processing applications on Intel IXP Network Processors, but NP-Click and μL/μC are much more focused on network data processing.

NP-Click and μL/μC require the programmer to specify the concurrency in the application by defining all data streams, whereas we make use of the COMPAAN compiler to do this work for the programmer automatically. Therefore, we are able to make use of an even more natural way of specification of the application, namely the sequential Matlab language.

Summarizing, using our work no new programming languages and methodologies need to be learned by the programmer. Programming is done using a simple imperative language, and all conversion is done automatically making use of COMPAAN.

1.4 Thesis Organization

First, we provide background information on the front-end of the IMCA tool in chapter 2. This includes the format of the sequential application, the COMPAAN compiler and the

parallel KPN specification. Next, we present the target platform in chapter 3. Details on the architecture and the programming model are provided. In chapter 4 is shown how to assign hardware resources to the elements of the parallel KPN specification. The next step, the generation of code for each hardware resource to implement the parallel KPN specification, is discussed in chapter 5. Chapter 6 deals with the process of handling incoming and outgoing data, as to provide an interface with the implemented KPN specification. Some experiments and results are provided in chapter 7. Suggestions for future work are discussed in chapter 8. Finally we provide conclusions in chapter 9.

1.5 Workload Partitioning

As this thesis is written by two individuals, we provide an overview of the partitioning of the total workload for the realisation of this thesis.

In the initial phase of the project, both authors familiarized themselves with the platform and with the programming of the platform. This required a fair amount of initial reading and experimenting with the Developer's Workbench. The implementation of the mapping and code generation is done by Johan Walters. Implementation of the process of receiving and transmitting data is done by David Snuijf. The design of both implemented parts is a result of cooperation and discussion between the two authors. As for writing the thesis, chapters 2 and 6 are the work of David, and chapters 3, 4 and 5 the work of Johan. The introduction, experiments and results, future work and conclusions are the shared results of both authors.

Chapter 2

Application Modeling

In this chapter we provide an introduction on the modelling of applications for our design flow as described in Figure 1.5 on page 6. It involves the specification of the initial sequential Matlab application and the conversion to a parallel specification by using the COMPAAN compiler. The sequential and the parallel specifications as well as the COMPAAN compiler are described, along with a transformation example.

2.1 Static Affine Nested Loop Programs

One restriction on using COMPAAN, is that applications have to be specified as static affine nested loop programs. This model is required to extract the data-level parallelism in the application, which cannot be computed otherwise. The specifications of this format is best described using an example as is shown in Figure 2.1.

```
01 int i,j;
02 matrix A;
03 for (i = 1, i < 5, i++) {
04   for (j = i, j < 12, j++) {
05     A[i,j] = 3*(i+j) - 3;
06     if (j > 3)
07               A[i,j+1] = i+j;
08   }
09 }
```

Figure 2.1: Example code describing Static Affine Nested Loops

In lines 5 – 9, all expressions are in the affine form of `ax+b`, where `x` can only consist of the variables specified in de loops wherein it is nested. All expressions in lines 5–9 are consecutively `i`, `j`, `3*(i+j) - 3`, `j`, `3`, `i`, `j+1` and `i+j`. All expressions match the format `ax+b` and are thus affine. All expressions only make use of variables specified by the loops they are in, thus form an affine nested loop program. Since the flow of the program can be determined at compile time, it is also static. For example, a statement like `if(a[i,j] == 0)` makes the control flow unpredictable at compile time. Such predictability is required for finding the exact dependencies between statements using Parametric Integer Linear Programming (PILP). Therefore, all conditions need to be known at compile time. The provided application is a so called *perfect* nested loop programs, as all statements are within the inner loop. If there were statements between line 3 and 4, it would have been an *imperfect* nested loop program. Using PILP, both versions can be handled.

Matching all criteria, the provided application is a static affine nested loop program and can thus be transformed using for the COMPAAN compiler.

2.2 Kahn Process Networks

A Kahn Process Network [12] (KPN) is a theoretic model of computation, consisting of a set of processes which communicate in a network through unidirectional communication channels. Processes can run concurrently as long as input data conditions are satisfied.

The processes communicate via unbounded FIFO[1] queues (of tokens). The processes use a blocking-read primitive on the FIFO queues to ensure correct synchronization between the processes. The FIFO queue channels are point-to-point, meaning that only one process can write to each channel, and only one process can read from it.

An example of a KPN network topology is shown in Figure 2.2. This example has three processes: A, B and C. The processes communicate using three FIFO channels: FIFO 1, FIFO 2 and FIFO 3. Process A can write to FIFO 1 and FIFO 2. Process B reads from FIFO 2 and writes to FIFO 3. Finally, process C reads from FIFO 1 and from FIFO 3.

Every process in a KPN is implemented as a sequential application, and executes concurrently with other processes. When a process reads from the queue, the process will suspend reading if the queue is empty, until there is enough data in the queue. This blocking-read characteristic leads to a deterministic behaviour of the model.

Writing to the queue is non-blocking, because the queue size is infinite in the theoretic model. However, in practical implementations of a KPN, unbounded FIFO queues are not possible. Therefore, these implementations also provide a blocking-write primitive. Whenever data has to be written to a queue which is full, the process will block until

[1] First-In-First-Out

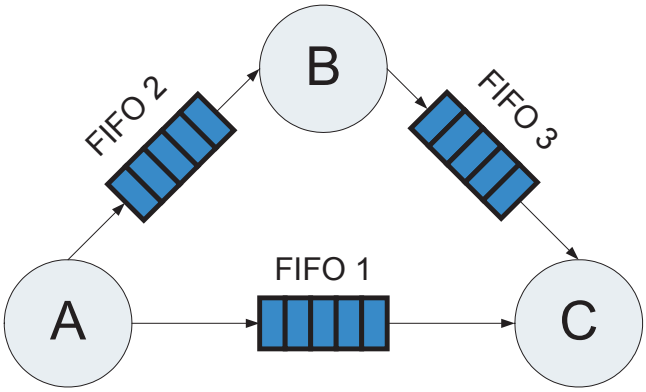

Figure 2.2: Simple Kahn Process Network

data is read from the queue by the process on the other side, after which the data can be successfully written and the process can resume. When the FIFO sizes are too small, this may lead to dead-lock behaviour. The minimum queue sizes has to be computed to avoid this situation [13]. If queue sizes are set sufficiently large, no blocking writes will occur.

Each FIFO channel is passive, and is controlled by the processes that perform operations on the FIFO channel. All FIFO channels are independent of each other.

The KPN model of computation is well suited for the application domain as the ones sketched in section 1.1, because of their streaming nature. A KPN model describes this streaming aspect well, as data is 'streamed' through the FIFO channels, being processed on the way. Because of the deterministic nature of the KPN model the output of the network is fully predictable. Lastly, the KPN model specifies an application in terms of distributed memory and distributed control. This enables us to map an application onto any multiprocessor platform systematically and efficiently.

2.3 The COMPAAN Compiler

The KPN model is well suited for mapping applications on a multiprocessor platform, as each process of the KPN can run on one processor of the multiprocessor platform. However, specifying an application using the KPN model by hand is very time consuming, difficult and error-prone. Luckily, the COMPAAN compiler translates sequentially specified applications into an equivalent KPN specification.

Leiden Embedded Research Centre [14], located in the Netherlands, developed a tool called the COMPAAN compiler. The COMPAAN compiler transforms DSP applications written in Matlab into Kahn Process Networks. Its framework consists of three tools, we'll explain step by step:

1. *MatParser*. This tool converts Matlab code into single assignment code (SAC). This will resemble the dependence graph (DG) of the initial nested loop program.

2. *DgParser*. This tool converts SAC into a Reduced Dependence Graph (PRDG) data structure. In terms of polyhedra this is a compact mathematical representation of the DG.

3. *Panda*. This tool converts the PRDG into a process network. It does so by generating a process for each node and a FIFO channel for each edge in the PRDG. Each parallel process will communicate with one another according to the data-dependency given in the DG.

The COMPAAN compiler transforms the Matlab application into a KPN specification in a fast and completely automatic manner. The applications that the COMPAAN compiler can transform correctly are parameterized static affine nested loop programs written in the Matlab language.

2.4 Example Transformation

In this section, we show how the COMPAAN compiler transforms the QRvr algorithm, specified as a sequential Matlab application, into an equivalent KPN specification. We choose QRvr because the program is not trivial and it is being used in the field of DSP. Another reason is that is not too complicated either.

The Matlab code for the QRvr is given in Figure 2.3 on page 16. It is written as an affine nested loop program. In lines 1 to 5, matrix R is filled with zeros. Lines 7 to 11 fill matrix X, using a function `Read()` which obtains data from the *source*. This function reads incoming data that is to be processed. Line 13 to 20 perform the actual algorithm using matrix R and matrix X as input for the cordic functions `Vectorize()` and `Rotate()`. Lines 22 to 26 pass the result in matrix R to the *sink*, i.e. the output. The functional correctness can easily be verified, but is outside the scope of this thesis.

Since the Matlab program in Figure 2.3 does not reveal the inherent data-level parallelism which is available in the application, we need to convert this sequential program into an executable parallel specification. For this we use the COMPAAN compiler in order to convert the program automatically into a the KPN specification.

Figure 2.4 on page 17 shows a schematic representation of the KPN specification of the QRvr algorithm as generated by COMPAAN. In this example, COMPAAN has created five processes, one for each assignment statement in the original Matlab program. In this case, these are calls to the functions `ReadMatrix_Zeros_64x64()`, `Read()`, `Vectorize()`, `Rotate()` and `Pass()`. The processes explicit the availability of the data-level parallelism in the application. The data dependencies and the communication between the processes are expressed in the KPN via the distributed FIFO channels.

ND_1 will execute the `ReadMatrix_Zeros_64x64()` function and send every element to processes ND_3 and ND_4. ND_2 will execute the `Read()` function and send elements to processes ND_3 and ND_4. ND_3 will execute the `Vectorize()` function and send the necessary output elements to itself, ND_4 and ND_5. ND_4 will execute the `Rotate()` function and send the necessary output elements to itself, ND_3 and ND_5. Finally the ND_5 will process the `Pass()` function and outputs the results as they would be stored in matrix R in the original Matlab code. ND_1 and ND_2 are called source nodes, as they provide data 'from outside'. ND_5 is called a sink node, as it outputs the data 'to outside'. Not shown in this representation is the order in which each process reads from incoming FIFO channels and outputs to outgoing FIFO channels. This information is stored as syntax trees in the XML formatted KPN file outputted by COMPAAN. Section 5.2 provides information on these syntax trees and the conversion of these syntax trees into program code.

```
%% Copyright (c) 2001 Leiden University (LIACS)
%% All rights reserved.
%%
%% Permission is hereby granted,  written agreement and without
%% license or royalty fees, to use, copy, modify, and distribute
%% this software and its documentation for any purpose, provided
%% that the above copyright notice appears in all copies of this
%% software. The software provided hereunder is on an "as is"
%% basis, and the copyright holder has no obligation to provide
%% maintenance, support, updates, enhancements, or modifications.
%%
%% $Id$
%% @author Edwin Rijpkema

 1  for j = 1:1:5,
 2    for i = j:1:5,
 3      [r(j,i)] = ReadMatrix_Zeros_64x64();
 4   end
 5  end
 6
 7  for k = 1:1:6,
 8    for j = 1:1:5,
 9      [x(k,j)] = Read();
10    end
11  end
12
13  for k = 1:1:6,
14    for j = 1:1:5,
15      [r(j,j), x(k,j), t ] = Vectorize( r(j,j), x(k,j) );
16      for i = j+1:1:5,
17        [r(j,i), x(k,i), t] = Rotate( r(j,i), x(k,i), t );
18      end
19    end
20  end
21
22  for j = 1:1:5,
23    for i = j:1:5,
24      [ Sink(j,i) ] = Pass( r(j,i) );
25    end
26  end
```

Figure 2.3: Matlab QRvr code

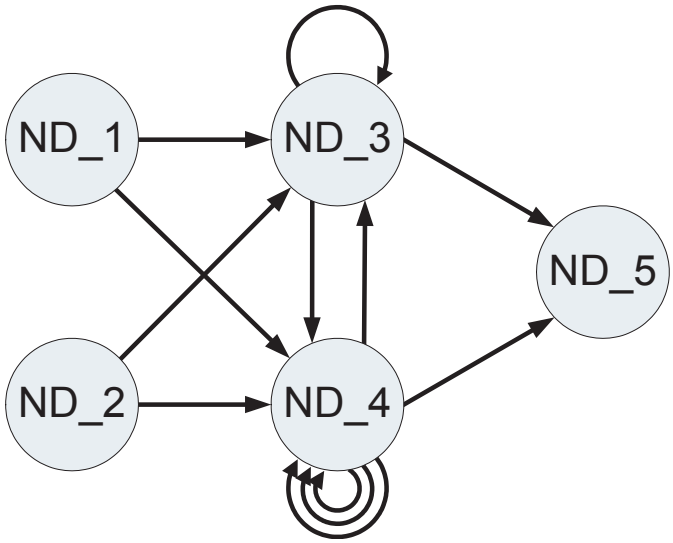

Figure 2.4: The KPN of QRvr

Chapter 3

The Intel IXP Network Processor

Intel has created over time a complete family of IXP network processors in the field of high performance computing. In this chapter, we first motivate our chose to use an IXP network processor. Then the Intel IXP2XXX network processor family is shortly discussed, and our choice for the IXP2400. We continue with a hardware architecture overview of the IXP2400. Programming the IXP, with all its pitfalls and difficulties, are discussed subsequently.

3.1 Choosing the Intel IXP Network Processor

The IXP2XXX network processor family is specifically designed and built to accommodate high speed network processing tasks. High throughput is achieved using fast gigabit interfaces and multiple parallel programmable processing units. The IXP is therefore well-suited for applications such as routing, network address translation and as a firewall. More complex functions can also be implemented, such as a VoIP[1]-server. Another common use is traffic analysis and monitoring. An IXP is fast enough to serve fairly large businesses for such functionality.

Clearly, the traditional application domain is network processing. Whatever data is sent over the network, some protocol is used for transportation. All networking protocols partition the data in packets. Numerous protocols are supported by the different IXP versions. Incoming packets are quickly stored into memory, after which the multiple processing units apply some clever function to the data. In router-like scenarios, calculation and modifications are done to the packet header(s). With traffic analysis, the packets are left intact. When a packet has been processed, it is ready to be sent out again.

From an streaming application point of view, such hardware is extremely interesting. Running a DSP application requires high-speed connections and multiple processors to

[1] Voice Over Internet Protocol

process the incoming data at real-time. The IXP precisely provides such a platform. Using the IXP platform for DSP applications moves the focus from calculation on the packet header to the payload of a packet. Widening the perspective of usage for the IXP would be beneficial both for Intel as for people in need for fast and affordable computing power. Exploring the potential of this new perspective of the IXP is one of the goals for this thesis, as is shown in chapter 7.

It seems incredible that the IXP is hardly being used for DSP related applications. A reason to account for this peculiarity is that the Intel's IXP range seems not to be well-known, as little references exist both on scientific work as on the internet. Another reason, probably maintaining this low fame, is that the IXP is pretty hard to program. More on this subject is covered in section 3.6. By building a tool facilitating easy program specification and automated mapping to the IXP platform, we hope to overcome these issues and provide a means for effectively using the hardware on a greater domain of applications than it is used for today.

3.2 The Intel IXP Network Processor Family

The Intel IXP product line of network processors include the IXP12XX, IXP4XX and IXP2XXX families. The older IXP12XXX range is replaced by the IXP2XXX series. The IXP4XX series do not include multiple processors and are thus not of interest for our research.

This leaves our focus to the IXP2XXX range, including the IXP2325, the IXP2350, the IXP2400, the IXP2805 and the IXP2855. The number of included processing units range from two to sixteen microengines, always plus one Intel® XScale™ core. Other differences are the supported memory sizes and protocols, and inclusion of a cryptographic unit. An overview of the differences in processing units and memory sizes is provided in Table 3.1.

Ultimately, our research is applicable to all members of the IXP2XXX family, as the hardware differences are easily parameterized. However, for building the tool we needed a hardware platform for generating test results. Also, one starting point is helpful for an initial programming effort. On LIACS [15], systems supporting the IXP2400 and IXP2855 are at our disposal. Having 8 microengines, using an IXP2400 system was sufficient for testing our research goals. This, along with practical reasons like other researchers using the available hardware, determined our choice for the IXP2400. As a result, we use the characteristics of the IXP2400 in this chapter to clarify the architecture of the network processor family in general.

Feature	IXP2325	IXP2400	IXP2855
XScale core	900 MHz max.	600 MHz max.	750 MHz max.
Microengines, organized into	2 at 600 MHz max. one cluster	8 at 600 MHz max. two clusters of 4	16 at 1.5 GHz max. two clusters of 8
SRAM	16 MB (1 channel)	128 MB (2 channels)	256 MB (4 channels)
DRAM	2 GB (2 channels)	1 GB (1 channel)	2 GB (3 channels)

Table 3.1: The Overview of the components of the IXP2325, IXP2400, and IXP2855 Processors

3.3 Architecture Overview of the IXP2400

To understand the operation of the IXP2400, it is needed to have knowledge of the architecture. It is moreover important to be able to program the hardware, as is discussed in section 3.6. Therefore, a relatively short overview of the IXP2400 is given in this section. A schematic block diagram overview of the most important functional units of the IXP2400 is visualized in Figure 3.1. The elements of interest for this thesis are discussed subsequently. Details like the many special status registers and some functional units are omitted in the figure as well as in this discussion. For further reading, see the Hardware Reference Manual of the IXP2400 [16].

3.3.1 Programmable Processing Units

Central to the design are the programmable processing units. The IXP2400 is equipped with one XScale core and eight microengines, all on the same die. The XScale core is a RISC[2] general-purpose processor, which is compliant with ARM [3] Architecture V5TE [17]. The microengines are RISC-processors which are optimized for typical fast-path packet processing tasks.

[2]Reduced Instruction Set Computer
[3]Advanced Risc Machines Ltd.

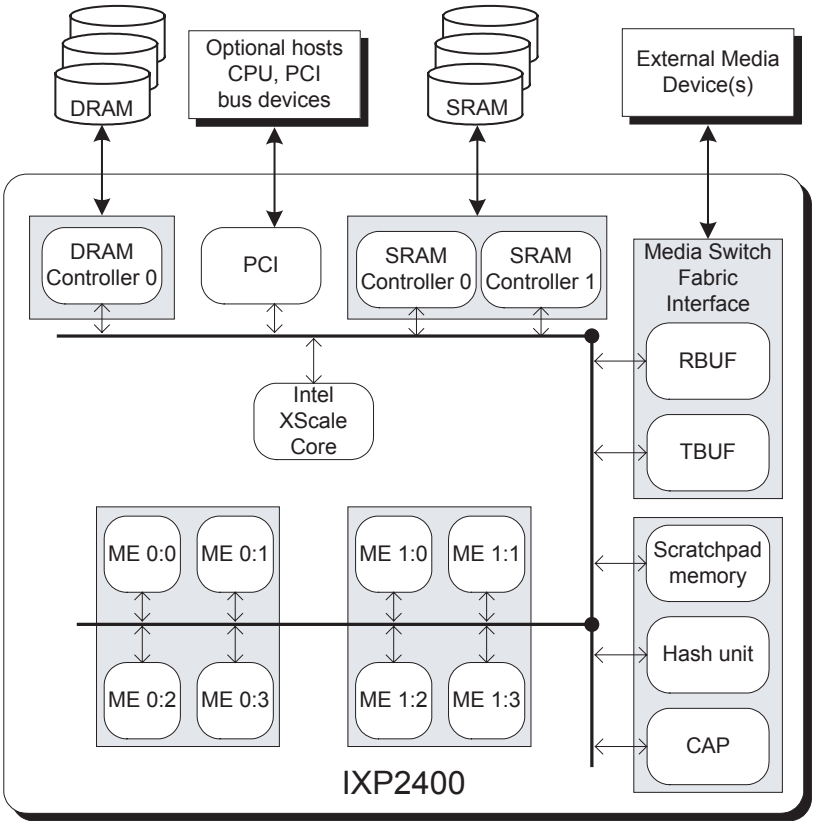

Figure 3.1: IXP2400

Intel XScale Core

The Intel XScale core is similar to the processing units found in many other hardware, including other embedded platforms, computers, handhelds and cell phones. The intended use on the network processors is controlling and supporting processes on the microengines, where needed. For example, if some exceptional packet is to be processed what would be troublesome to implement in the nicely streamlined packet flow designed on the microengines, the XScale core can handle that packet. The XScale core is, in contrast to the microengines, easy to program. Lots of tools, compilers and operating systems are available for the Intel XScale processor. For the scope of this thesis, we found no special use for the XScale core.

Microengines

The instruction sets in the microengines are specifically tuned for processing network data. It consists of over 50 different instructions including arithmetic and logical opera-

tions that operate at bit, byte, and long-word levels, and can be combined with shift and rotate operations in a single instruction. Integer multiplication is supported, but floating point operations are not. The pipeline consists of six stages in which instructions take on average one clock cycle to execute. Figure 3.2 on page 23 shows the functional blocks in each microengine. On the top side of the figure, registers are used to store incoming data. These are next-neighbour registers, DRAM read transfer registers and SRAM read transfer registers. The read transfer registers store data from SRAM memory and from DRAM memory. On the bottom side of the figure, DRAM write transfer registers and SRAM write transfer registers are used to move data out of the microengine, to DRAM memory and to SRAM memory. The third data output option is to the next-neighbour register bank of the next microengine in line. There are two general purpose register banks. The register types are discussed in more detail in section 3.3.3. Other elements of interest are the instruction store with a size of 4 K instructions, the local memory with a size of 640 longwords of data and the execution datapath.

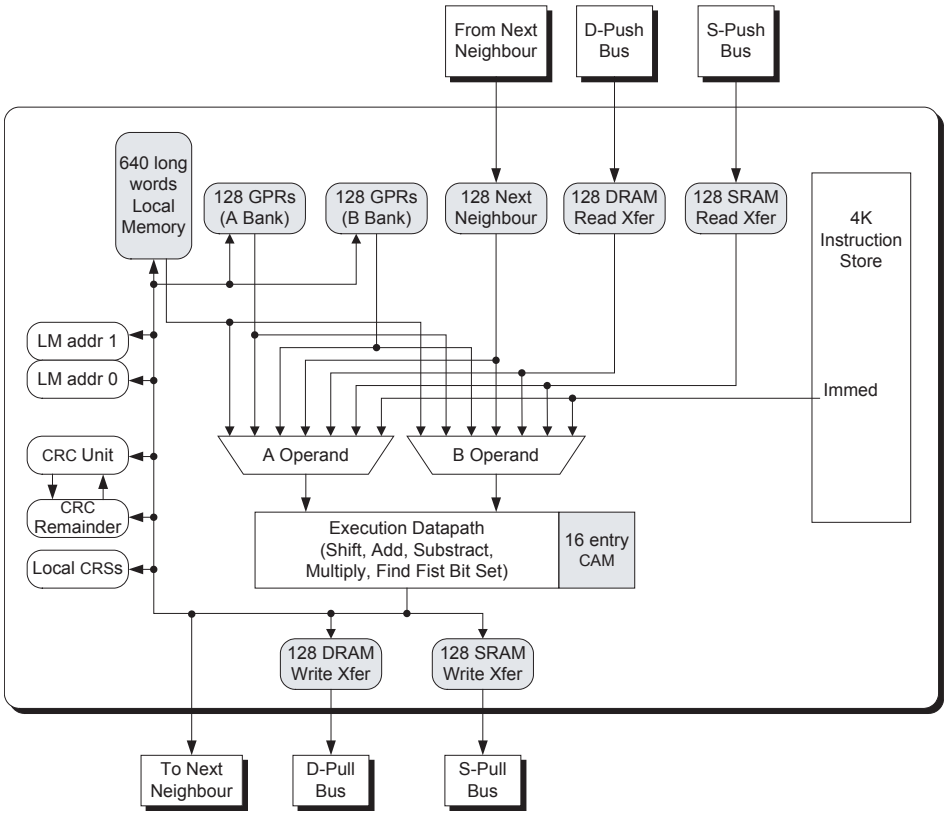

Figure 3.2: IXP2400 Microengine Block Diagram

3.3.2 Threads and Thread Arbitration

An important feature of the microengines is hardware-assisted thread execution. Each microengine has 8 threads. Switching contexts between threads cost no more overhead than aborting an execution due to a branch instruction, 4 clock cycles on average. This is possible by each of the eight threads per microengine having its own set of registers and program counters. Thus, instead of storing all register contents of the latest running thread into memory and retrieve the register contents for the next one, all taking hundreds of cycles, simply a different set of registers is activated. Threads can quickly switch to another thread on the same microengine, within 4 clock cycles. Whenever a thread needs to wait for data, the next thread immediately takes over. This way, by executing code of other threads while waiting for data, memory latencies are hidden.

The IXP2400 processor also contains a thread arbiter that swaps between threads in a microengine in round-robin order. Only threads that are ready to run, e.g. threads no longer waiting for data to arrive, are activated. When a thread is waiting for data, the ready state will be set as soon as the data has arrived. The round-robin schedule of 4 running threads is illustrated in Figure 3.3. It is clear that two threads cannot run at the same time.

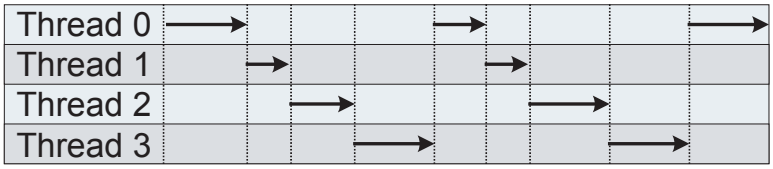

Figure 3.3: Threads

Each thread has its own register set and program counter. Registers can also be set to be shared between all contexts of the microengine. The 4 K 40-bit instruction store is always shared between contexts. This means that all threads execute the same code. However, code can branch based on the thread number of the current executing context, thus allowing threads to execute their ’own’ code. Still, 4 K is not much to share.

Cooperative Threading Example

Consider the following packet-processing scenario to illustrate the purpose of the design of threads. Suppose a packet stream is recorded on a place in memory (visible to all microengines). Further, a specific task has to be applied to all packets, and this task is to be done by one microengine. The algorithm for the task fits in the total instruction store of the microengine. Now, all threads take turns in executing the task on subsequent packets, each thread doing the next packet in line, one at a time. An example implementation of this process is illustrated in the pseudo-code in Figure 3.4.

```
 1  int (shared)  packetCounter = 0;
 2  int (private) myPacketCounter;
 3
 4  main () {
 5    myPacketCounter = packetCounter;
 6    packetCounter ++;
 7
 8    data = getDataFromMemory(myPacketCounter);
 9
10    swap out;
11    wait for data to arrive;    // if not arrived, swap out again
12
12    applyTask(data);
14  }
```

Figure 3.4: Cooperative Threads Pseudocode

Every thread uses it's own copy of the packet counter (line 5). To hide latencies, whenever a thread swaps out it will still remember what packet it was working on, while other threads can continue working on subsequent packets.

When the program starts running the first thread will process packet number 0. In line 8, it must first retrieve the data from memory. Therefore, it will swap out on line 10. At that moment, the second thread will start processing packet number 1. By the time the first thread is allowed to run by the round-robin schedule of the thread arbiter, depending on whether the data has arrived, it will continue processing packet number 0. Meanwhile other threads are waiting for their data to arrive. In this way eight packets can be processed in parallel with as little memory latencies possible. That the two variables are declared private or shared is essential for the correct working with multiple threads.

The threading model is non-preemptive, meaning that the thread will only swap out if the code tells it to do so. For a programmer, it is important to keep control over the order in which things happen. For example, no context switch will be done between lines 5 and 8. If the thread would swap out between lines 5 and 6, two threads would end up processing the same packet. By not placing any code that could cause switching of the context provides a form of mutual exclusion. This implies that no other threads will see chance to execute. The programmer must manually ensure that context switching occasionally occurs to give all threads opportunity to run.

3.3.3 Registers

Each microengine has four different types of registers: general purpose (GP), SRAM transfer, DRAM transfer and next-neighbour registers. Microengines do not need to be flushed into memory when the context switches. An equal portion of the registers is assigned to each thread. Therefore a context switch can be done within 4 clock cycles.

Each microengine has 256 32-bit GP registers, separated in two banks of 128. These can either be used for thread-local communication (the equal share portion) or can be shared between all contexts for inter-thread communication For example, this means that when no GP registers are shared, each thread will have 16 private GP registers.

Each microengine has 256 SRAM transfer registers, divided into 128 SRAM read registers and 128 SRAM write registers. These registers are used to read from and write to all functional units on the network processor except for the DRAM memory. Therefore these registers form the main means of communication for the microengine. When data stored in SRAM is to be read, it is first placed in a SRAM read register. Once data arrives there, it can be used in the microengine. When data needs to be stored into SRAM, it first has to be placed in an SRAM write register after which it can be stored into SRAM.

An equal amount of DRAM transfer registers is present on each microengine and is organized in the same fashion as SRAM transfer registers. While the DRAM read registers can be used to read from all hardware on the network processor, the write registers can only be used for DRAM memory access.

The last type of registers are the next-neighbour registers, of which each microengine has 128 of 32-bit. These can be used to send data to the next microengine in number. The first microengine cannot read data from a previous microengine, and the 8th microengine cannot write data to a next. All 128 registers can be used as one big hardware assisted ring with atomic get and put operations, or be partitioned between the threads. Lastly it can also be used as a set of slower extra GP registers, instead of using it as inter-thread communication. Any data put in the registers can then only be read by the same microengine.

3.3.4 Memory

Except for the registers, memory is also present on each microengine. Most important is the local memory, with a size of 640 long-words. It has a latency of 3 cycles. Another form of memory local to the microengine is the content addressable memory, which is a lookup table consisting of 16 entries.

Still on the network processor chip, but outside the microengines, memory is present called *scratchpad memory*. It has a size of 64 K 32-bit values. It is has longer access latency times than the local memory, but still is much faster than the off-chip memory. Scratchpad and the off-chip memory are shared between all microengines. It supports atomic operations like add and subtract, and 16 FIFO rings with atomic get and put op-

erations. These rings are implemented using special hardware registers functioning as base, head and tail pointers. The content of the rings are on the scratchpad memory itself.

Outside the network processor chip, SRAM and DRAM memory banks are present. Up to 1 GB of DRAM and up to 128 MB of SRAM are supported. These memory banks are accessed using the SRAM and DRAM transfer registers. SRAM also offers atomic operations and FIFO rings.

Please note that the hardware supported rings on scratchpad and in SRAM memory, as well as the next neighbour ring, are FIFO implementations. This is an important clue for mapping FIFO specifications from a KPN onto the platform. It is possible to implement more FIFO buffers in the various memory types than there are rings available in the scratchpad memory and SRAM memory, albeit slower because the get and put operations will not be atomic. The mapping of FIFO channels onto the memory hardware is discussed in section 4.2.2.

An overview of the properties of the memory types is provided in Table 3.2 on page 27.

Memory	**Logical width (in bytes)**	**Size (in bytes)**	**Approx. unloaded latency (in clks)**	**Special operations**
Local memory	4	2560 (per microengine)	3	Indexed addressing with post increment and decrement.
Scratchpad	4	16K (on-chip)	60	Atomic operations including atomic subtract. 16 rings, with atomic get and put operations.
SRAM (QDR)	4	128M (addressable, 64M per channel)	90	Atomic operations, excluding atomic subtract. 64-element queue array with atomic enqueue, dequeue, get and put operations.
DRAM (Rambus/ DDR)	8	1G (addressable)	120	Direct path to and from the MSF, which allows data to be moved between the two without first going through one of the processors.

Table 3.2: Properties of the four IXP2400 Processor Memories

3.4 Media and Switch Fabric Interface

For communication with the outside world, two hardware units are present: the PCI controller and the Media and Switch Fabric Interface (MSF). The PCI controller connects to nearby hardware, where the MSF connects to a physical layer device and/or a Switch Fabric Interface (in other words: the network).

Packet reception and transmission on the IXP2400 using the MSF is a complex act of segmenting and reassembling small partial-packet data, called mpackets. Basically, the MSF contains two buffers for mpacket storage, one for inbound data (RBUF) and one for outbound data (TBUF). An mpacket is configurable to be either 64, 128 or 256 bytes of size.

Incoming packets are partitioned into mpacket size, and stored into RBUF. To be able to restore the original packets, Start of Packet (SOP) and End of Packet (EOP) flags can be set per mpacket. The presence of an mpacket is recorded on a waiting list.

At least one thread of one microengine should be assigned to handle incoming data on the MSF. It does so by placing its thread number on another waiting list on the MSF. When both an mpacket and a waiting thread are available, the mpacket is assigned to that thread[4]. The thread can now either read the mpacket data itself, or instruct the MSF that the data is to be stored to some address in DRAM, as there exists a direct link between the MSF and DRAM. Packets are reassembled in DRAM this way. Using a single thread to do this work, is in practice too slow. Typical IXP applications use all eight threads on one microengine, and so does our implementation.

The other way around, threads can instruct the MSF to accept mpacket-sized data stored in DRAM into the TBUF of the MSF. SOP and EOP flags need to be set at the first and the last mpacket, respectively. The MSF will reassemble and transmit the packet onto the network.

This might all sound confusing. To put it more simply, threads assist the MSF in storing incoming packets into DRAM, and transmit outbound packets stored in DRAM.

3.5 Signals

One more important feature of the IXP2400 is the unified signalling model. Signals can be generated by almost all parts of the hardware. Microengines can test for presence of absence of certain signals. This allows for conditional branching upon presence or absence of a signal, or to tell the thread arbiter not to swap until a certain signal is raised. It is also possible to have multiple outstanding signals at the same time, even at the same hardware. Many of the processes running on the IXP2400, are controlled using signals.

[4] Actually, when either a thread or an mpacket is already waiting, and the other arrives, the latter is not put on its waiting list, but assigned directly.

```
01 __declspec(sram_read_reg) x; SIGNAL sig;
02 __declspec(scratch)* addr = 0x400;
03
04 main () {
05    scratch_read(&x, addr, 1, sig_done, sig);
06    do_other_work();
07    __wait_for_all(&sig);
08    y = x;
09 }
```

Figure 3.5: Signal to wait for memory operation

A code example of the use of signals is given in Figure 3.5. In line 5, a memory operation is issued. The contents of memory address $0x400$ in scratchpad memory are transferred to the SRAM read register `x`. The command `sig_done` indicates that the microengine can continue processing. Line 6 will be executed directly after line 5. Line 7 indicates that the thread may only continue after the memory operation completed. When the memory operation of line 5 completes, a signal `sig` is raised. This signal is what line 7 checks for. Between line 5 and 7 no use of the variable `x` can be made. After line 7, `x` will be guaranteed to contain the value read from scratchpad memory. The option of executing code between the issuing and the completion of the memory operation allows is a form of *hiding* the memory latency.

```
   // Microengine 1                      // Microengine 2

01 __declspec(remote) SIGNAL sig1; 10 __declspec(visible)SIGNAL sig1;
02 __declspec(visible)SIGNAL sig2; 11 __declspec(remote) SIGNAL sig2;
03                                 12
04 main () {                       13 main () {
05     signal_next_ME(sig1);       14
06                                 15    __wait_for_all(&sig1);
07                                 16    signal_previous_ME(sig2);
08     __wait_for_all(&sig2);      17
09 }                               18 }
```

Figure 3.6: Synchronisation between microengines using signals

Another code example of the use of signals is given in Figure 3.6. Lines 1 – 9 are executed on microengine 1, lines 10 – 18 on microengine 2. In line 5, a signal is sent to microengine 2. In line 15, microengine 2 waits for that signal to arrive. In this example microengine 2 sends a signal back to microengine 1, in lines 16 and 8. This way, the microengines are synchronized to each other. In our work, this type of synchronisation is used after initialization of the FIFO structures. All microengines are synchronized with each other, such that each microengine is guaranteed that all FIFO structures are initialized properly. Another use is described in section 6.5.

3.6 Programming the IXP

The programming model of the IXP2400 processor differs from traditional programming models, such as those used on Windows or Linux-based environments. In those traditional programming models, most hardware aspects of the platform are virtualized, abstracting the concepts of threads, memory and signals to the programmer. Such a virtualization was not developed for the IXP programming model.

Instead, programmers need to have understanding of all the hardware concepts as most are controlled directly by the programming language. There are no memory caches. When used improperly, memory accesses will cause corrupted data usage. For each variable, the programmer has to specify (and remember) in what memory or which register it resides, whether it is shared between threads or not and whether it is used by multiple microengines or not. For each memory operation, signals are generated and have to be handled properly. Memory access latencies have to be hidden as efficient as possible. Thread swapping is not done automatically, but has to be enforced by the programmer. Synchronization between concurrent processes in the application needs to be modelled manually. The many special status registers need to be used correctly with specific bitpatterns.

The lack of virtualization and abstraction is by design, as it was felt that complete understanding of the hardware is needed to produce optimal code. Otherwise, it would supposedly not be possible to let the hardware fully do what it was designed for. True, writing optimal code is possible, however the process is time-consuming, error prone and difficult.

3.6.1 Programming Languages and Environment

Intel provides several programming tools combined in the Intel IXA Software Development Kit. These tools include an assembler for the microengine assembly language, a compiler for the microengine C language, as well as an integrated desktop environment (IDE) called the Developer's Workbench.

Microengine Assembly

The microengines have a rich instruction set, designed to support operations particularly useful for networking applications. Therefore, an assembly language to support the instruction set, but also all other hardware on the IXP, was developed. As with most assembly languages, it provides means of creating low level code as optimal as possible, but is not necessarily easy to program.

Microengine C

Intel also created a C variant for programming the IXP called the *microengine C language*. The syntax is ANSI C, with the exception of no support for function pointers and recursion. Type safety, pointers to memory, functions, enums, structures and arrays are supported. Because of the many exposed memory and register types, variable declarations usually include an extra modifier, `__declspec`, to inform the compiler where to store the given variable. For example, `__declspec(scratch) int x` defines an integer variable `x` stored in scratchpad memory.

Not all features of the IXP2400 hardware can be expressed in ANSI C, such as asynchronous memory access, waiting for signals and special memory operations. Therefore, a library of intrinsics is provided. In essence, the compiler replaces an intrinsic function call with well-known assembly code, implementing the special operation. For example, to read a value from scratchpad memory and increase the value afterward in one atomic operation, the intrinsic `scratch_test_and_incr()` is provided.

Developer's Workbench

With the Developer's Workbench, it is possible to develop and debug programs using both the microengine assembly language as the microengine C language. It provides a graphical environment running on Microsoft Windows with a syntax-highlighting editor. The assembler, microengine C compiler and linker are controlled from the Developer's Workbench.

Included is a simulator to emulate the IXP2XXX hardware. The simulator also provides a packet stream generator to provide incoming data. Using the simulator, it is possible to debug every process on the platform, as every hardware detail is simulated. All 'current' data in all memories and registers can be watched, as well as variables. Per thread, the current executed instruction is given. Breakpoints can be set per line of code and per thread. Breakpoints can also be set on changes in memory locations and variables.

To support the debugging process further, a history collector can store all memory and variable data per executed clock cycle, as well as the point of execution per thread. An annotated graphical representation of the thread's execution history is also provided.

Despite the numerous debugging tools, debugging is still a painful task. This is partly due to the complexity of the hardware which remains hard to oversee, but mostly because the compiler optimizes as much code away as possible. Sometimes, this results in not being able to see the contents of variables that are optimized away. Also variables often appear to be out of scope even in the lines where they are actually used. When the programmer wants to see the contents of such variables, it is needed to add extra code that will not be optimized away to be able to see the contents. An even worse, a side effect of the optimization by the compiler is that sometimes lines of code will not be executed at all. For example, when a variable is written of which the compiler thinks it is not used afterwards. In such cases the programmer must add an `implicit_read` statement for these variables to avoid being optimized away.

The simulator is slow, as it has to emulate and store every state of the hardware per clock cycle. Debugging using hardware does not suffer from the simulator's slowness, but provides much less detail.

Chapter 4

Mapping

The conversion from a KPN specification to an IXP Network Processor implementation consists of two stages. First, all elements of the KPN have to be mapped onto the hardware resources of the platform. Second, code has to be generated for each hardware resource. The latter is described in chapter 5.

In this chapter we propose several techniques to map a KPN specification onto the IXP2400 platform. The elements of the KPN that are to be mapped, are processes and FIFO queues. The mapping target is the hardware described in chapter 3.

4.1 Process Mapping

4.1.1 Processes to Processors

When mapping process networks to multiprocessor platforms, processes are generally mapped onto processors. An exception to this is the LAURA[3] tool, as it uses reconfigurable logic on FPGA boards for processing power, and not necessarily the embedded general purpose processors if these are available. This comparison is somewhat insipid, as FPGA hardware is not always a multiprocessor platform. For example, microprocessors can be implemented in reconfigurable logic and general purpose processors can be embedded in the hardware, but both can be absent. In the context of multiprocessor platforms a comparison to the ESPAM[4] tool is more valid, as it implements MicroBlaze [18] processors and the interconnects onto the FPGA logic, which renders the FPGA a multiprocessor platform.

In the case of the IXP2400, the only processing power is available in the microengines and the XScale core. Processes from the KPN will be mapped onto processors. The IMCA tool does not map processes onto the XScale core, as it is a completely different type of processor than the microengines. It lacks the threading model and has a different communication structure with the other hardware on the platform. We found

it infeasible to implement a completely different mapping solution just for this single processor as there are many more microengines. Therefore, we leave this matter for future implementations (see in chapter 8).

The microengines on the IXP2400 have a special hardware assisted threading model. The remainder of this section discusses how to map processes effectively using these threads. Therefore we first discuss the subject of cardinality in the mapping process.

4.1.2 Cardinality

Input to the mapping process is a KPN with a certain amount of processes, and a platform with a certain fixed amount of processors. The amount of processes in the KPN is variable to a certain degree. Using algorithmic transformation techniques [19] different nodes of a KPN can be merged into a single node, or the workload of a single node can be split into two nodes, with half the workload. This allows for a variable number of nodes ranging from one (all nodes merged together) to an amount related to the number of loop iterations available in an application (all loops unrolled or unfolded). Thus, COMPAAN can generate a less, an equal or a greater amount of processes than the number of processors on the platform, depending on the application. Each situation would require a different cardinality for the mapping strategy, i.e. the relation between the amount of processes and the processors to map the processes onto.

Choosing the cardinality of the mapping of KPN processes onto processing units is important for both guaranteeing the deterministic behaviour of the application, as well as the performance of the mapping result. Most intuitive is a one-to-one mapping, but many-to-one and one-to-many should be considered as well.

- A one-to-one mapping means that each process is mapped onto one processor, and that on each processor one process has been mapped.
- With a many-to-one mapping, one or more processes are mapped onto each processor (but not limiting the amount of processors to one).
- A one-to-many mapping means that duplicates of one process are mapped onto multiple processors on the platform.
- A many-to-many mapping implies that duplicates of processes are allowed, and multiple processes on each processor as well.

First, let us rule out one-to-many and many-to-many mapping of processes onto processes: *one process should be mapped only once to a processor*. This is an important aspect of the mapping process to not disturb the correct working of a process network. If one process were to be mapped on two different processors, correct deterministic behaviour would be no longer guaranteed. The problem lies in how to split the corresponding FIFO channels. The interesting part is that COMPAAN can split one process into

two processes, creating the FIFO channels correctly. Therefore, the mapper should not undertake such efforts leaving the splitting process to COMPAAN and map one process only once.

This leaves us to one-to-one and many-to-one mapping. The question is how many processes we should let COMPAAN generate for our platform, one for each processor or multiple for each processor? If there were enough processors available to match the number of possible parallelizations in an application, the maximum amount of performance increase, due to parallelization, is reached. In practice, the amount of processors is limited to a smaller amount, thus limiting the amount of tasks that can be truly performed concurrently. If COMPAAN would create more processes than processors by splitting one existing process into two, these two new processes would be mapped onto the same processor again (if the other processes would be mapped onto the same microengines as they were before, i.e. ceteris paribus). If the latencies of accessing the FIFO channels and swapping between processes were neglected, the situation will stay the same.

The presence of these latencies implies that mapping multiple processes onto a single processor cause a loss of performance. However, we will show that in the case of the IXP2400 this also has beneficial aspects.

4.1.3 Latencies

In the general case of a multiprocessor platform, two types of latencies are to be considered on each processor: swapping latency and memory access latency[1]. Depending on these latencies, it can be decided whether it is wise to map multiple processes onto one processor or not.

Swapping Latency

Most processors suffer from a latency when switching from one task or thread to another. All register contents have to be saved into memory, and a new set of contents have to be loaded into the registers. When swapping latencies are costly, it forms a penalty for mapping multiple processes onto a single processor. In the case of the IXP2400, swapping latencies are negligible due to the hardware assisted threading model. As far as swapping latencies are concerned, mapping a process to each of the eight threads of the microengine is as costly as mapping only one process to the whole microengine.

[1] In general purpose processors swapping latencies are partially caused by memory access latencies. For this discussion a distinction is necessary.

Memory Latency

Considering everything outside the processor as 'memory', the latencies for accessing these memories are also an important factor. FIFO channels are also considered memory. Every process communicates through FIFO channels, and thus suffers from latencies of accessing the hardware outside the processor. One strategy to keep this penalty to a minimum is to minimize the number of FIFO channels that have to be accessed on a single processor. This is achieved by not letting COMPAAN split any node into multiple nodes. Less duplicated processes means less FIFO channels, and thus the number of memory access latencies.

A better strategy is to *hide* the memory access latencies altogether. This is exactly what the IXP2400 was designed to do, as described in section 3.3.2. Every time the microengine needs to access a FIFO channel, another task can take over the microengine. The more processes mapped on the microengine, the bigger the chance that another process is ready to run. This way, all memory accesses can be hidden as much as possible. If all memory accesses are truly hidden, a better result is achieved than when using only as little memory accesses as possible.

There is one downside: the microengines on the IXP2400 only have a limited amount of instruction store, so it may not be possible to map any eight processes onto one microengine, depending on the code size. Parallelizations obtained via loop unrolling have this problem in less extend as the corresponding processes share the same function code and can be mapped onto the same microengine without causing a large increase of code size.

A remark is that in our mapping implementation, *all* FIFO channels are mapped onto hardware outside the processors. When mapping FIFO channels that connect processes mapped onto a single microengine *inside* the microengines, e.g. in the local memory, memory latencies would further be reduced.

4.1.4 Processes to Threads

Concluding from the previous section, a many-to-one mapping is best practice on the IXP2400. When mapping more processes onto each microengine, memory latencies are better hidden. Each microengine has eight threads. In our mapping implementation, five of the eight microengines are available for mapping processes onto. The other three microengines are used for transmitting and receiving data. This means that in theory $5 \times 8 = 40$ processes can be mapped onto an IXP2400. However, if each process requires an algorithm with a large code size, mapping 8 functions would not fit in the 4 K instruction store, which is shared by all threads. In practice, most process functions are quite small and together easily fit the instruction store. Per microengine about 3K instruction store is available for process algorithms, as 1K is used for implementing the KPN syntax trees and FIFO queues.

We have used the term 'many-to-one mapping' for mapping multiple processes onto each microengine. It is also valid to say we use a one-to-one mapping of processes to threads. Each thread could be seen as a separate processor, as it has its own set of registers. This would mean we have a one-to-one mapping of processes to processors after all. We will not adopt this terminology of naming threads as separate processors, as it is confusing.

4.1.5 Resources

Having mapped a process to a thread, what resources are available to the process? Included are at least an equal share of the registers of the microengine for each thread. This is default behaviour for the IXP2400. A portion of instruction store is also used to implement the algorithm of the process. A set of registers may not be sufficient storage space. It is therefore logical to allow usage of local memory by each thread, as it has a short access time.

Usage of scratchpad memory, SRAM memory and DRAM memory should be disallowed, as it would harm the streaming character of the application by introducing new memory access latencies aside from the required FIFO queue accesses.

Another reason only to use registers and local memory is that programming for the IXP2400 then is almost as easy as regular C programming. Usage of hardware outside the microengines makes programming much harder by needing special programming constructs specific to the IXP, such as signals and intrinsic functions. Process algorithms are usually not included in the KPN specification, so they still have to be implemented. The IMCA tool will generate an empty function for the programmer to implement.

4.1.6 Strategies

The question of how many processes should be mapped onto the threads is closely related to *where* each process should be mapped. There is no difference between the threads of the microengines, so when mapping a process to a thread of a microengine, it does not matter which thread will be assigned. Furthermore, the only difference between the microengines is the topology of the next-neighbour communication channels.

Based on this information, three distinctions can be made when considering mapping a process to a specific thread:

1. Which processes are mapped onto the previous microengine?

2. Which processes are mapped onto the next microengine?

3. Which processes are mapped onto the same microengine?

The first two distinctions are related to the next-neighbour communication channels. When mapping FIFO channels onto these next-neighbour channels, receiving processes should be mapped on the next neighbour microengine of the microengine of the sending processes. Our mapping implementation does not use next-neighbour communication channels, but it should be implemented in the future as described in chapter 8. More on the subject of mapping FIFO channels is found in section 4.2.

The third distinction is the most important one. It is of significance for three reasons. First, the total load of each microengine (the combined execution time needed by the processes) should be kept as equal as possible. To spread the load, it is important to consider which processes should be mapped together onto the same microengine. A simple heuristic to implement this strategy, is to assume that every process has an equal load, thus each microengine should have an equal number of process mapped onto it. To do so, first map one process onto each microengine, then a second, and so on – until there are no more processes to map.

The second reason is the matter of code size. If several processes use the same function code, it is wise to map these processes onto the same microengine to save total code size.

The third reason is that FIFO channels between processes can be implemented inside the memory of a microengine. When mapping processes with a lot of mutual FIFO channels onto the same processor, communication overhead (memory latency) is saved. This heuristic may conflict with the heuristic of load balancing, as the processes with the most mutual FIFO channels often are the processes with the biggest load.

Due to the period of time available for this thesis, only the described heuristic of load balancing is implemented in the mapper for our IMCA tool. An example of this mapping strategy is illustrated in Table 4.1. It represents the mapping of the FDWT algorithm, of which the corresponding process network is illustrated in Figure 4.1 on page 45. Microengine 0:0, microengine 0:1 and microengine 1:3 are reserved for I/O operations as described in section 6. Nodes are mapped in a round-robin order onto the available microengines. The mapping strategy is once more visualized in Figure 4.2 on page 46.

4.2 Channel Mapping

After mapping the processes of the KPN to the threads of the microengines, the next step is mapping the KPN FIFO channels onto the IXP2400 hardware. We assume that the tokens used for these channels are 32 bit integer values.

	Microengine							
Thread	**0:0**	**0:1**	**0:2**	**0:3**	**1:0**	**1:1**	**1:2**	**1:3**
1	RX	read pkt	Node 1	Node 2	Node 3	Node 4	Node 5	TX
2	RX	read pkt	Node 6	Node 7	Node 8	Node 9	Node 10	TX
3	RX	read pkt	Node 11	Node 12	Node 13	Node 14	Node 15	TX
4	RX	read pkt	Node 16	Node 17	Node 18	Node 19	Node 20	TX
5	RX	write pkt	Node 21	Node 22	Node 23	-	-	TX
6	RX	write pkt	-	-	-	-	-	TX
7	RX	write pkt	-	-	-	-	-	TX
8	RX	write pkt	-	-	-	-	-	TX

Table 4.1: Mapping of nodes from Figure 4.1 onto IXP2400

4.2.1 FIFO Characteristics

In essence any FIFO queue implementation is an amount of memory with special characteristics with regard to how data is written into the memory and is retrieved from the memory. The most important aspect of any FIFO queue, as the name suggests, that the data that was first written, will be the first to be send out. To ensure this behaviour, FIFO implementations typically support two special operations: `get()` and `put()`. The implementation of the FIFO queue should ensure that the elements inside are kept sorted, and that these functions operate on the correct elements inside the queue. A common implementation is that two pointer values are maintained, containing the memory addresses of the first free element to read (the *head*) and the first element to write at (the *tail*).

When multiple instances write to or read from the same port[2], the `get()` and `put()` functions need to be atomic operations. It may not happen that two functions use the `put()` operation, yet both values are written to the same element inside the queue as a result of using twice the same head pointer value. Thus, writing a new element and raising the head pointer should be performed atomically to avoid race conditions.

When only one instance can read from or write to a single FIFO queue, a race condition caused by multiple instances trying to use the same operation will not occur, even when the operations are not atomic. To avoid a race condition between the writing and reading instances, the operations implementing `get()` and `put()` should be executed in the correct order. For example, for a `put()` operation the order will be: *read* the head value, *write* to that position, *raise* the head value.

A KPN uses point-to-point FIFO channels. Every FIFO channel can be written by only one process, and only one process can read from it. Using a one-to-one mapping of

[2]This is not the case for point-to-point FIFO usage as with the KPN model.

processes to threads on the microengines, each FIFO channel should also be mapped once. The KPN topology is kept intact by the mapping and the point-to-point aspects are maintained. Ergo: for the mapping, atomic `get()` and `put()` operations are *not* required.

4.2.2 Hardware Options

The IXP Network Processors were created for streaming applications. In such applications, FIFO queues are very common. It is not by accident that several structures implementing FIFO queues are available in hardware. Intel calls these hardware FIFO implementations *rings*. Rings available in the IXP2400 are next-neighbour rings, scratchpad rings and SRAM rings. These rings all support atomic `get()` and `put()` operations, by using dedicated head and tail pointer registers for each ring. These registers are also called *ring descriptors*. Next-neighbour rings have the shortest access times, SRAM rings the longest. Sadly, there are only 7 next-neighbour rings (because there are 8 microengines on the IXP2400) and there are only ring descriptor registers available for 16 scratchpad rings. For SRAM rings there are 64 ring descriptors available per SRAM controller, of which ring descriptor values can be cached into SRAM itself. This means that the number of SRAM rings is only limited by the amount of SRAM memory available.

As we are not specifically interested in the atomic operations but merely in fast access times, more FIFO implementations were thought of. These include the use of local memory, reflect-write operations (microengines directly writing data into each others transfer registers), scratchpad memory without ring support, and DRAM memory. The idea is to assign two memory addresses as head and tail pointers and a series of memory addresses as queue. Interpretation and updating of the memory addresses of the head and tail pointers are implemented on the microengine instead of in the memory ring hardware. Since all FIFO channels are point-to-point, the status of the FIFO channels can be cached in the microengines. This saves memory accesses when it is already known from a previous access that there is plenty room to put tokens into, or there are enough tokens to read.

Each hardware mapping option is described subsequently in order of memory access times. These access times are based on Table 3.2 on page 27. Our mapping implementation supports both the hardware and software supported scratch rings, and SRAM rings. Implementations of DRAM FIFO queues would be too slow. Use of Next-neighbour rings, reflect-write operations and local memory is left for future implementations. A topological overview of the various implementations on the hardware is provided in Figure 4.3 on page 47.

Local Memory Rings

Each microengine has a fast accessible local memory of 640 longwords that is shared between all threads. This makes this memory an ideal candidate for usage as FIFO queue for all channels that have a single process both as source as destination, and channels between processes mapped onto threads of the same microengine. Figure 4.3 shows local memory FIFO s as number 1.

The local memory cannot be exclusively used for FIFO queues. Depending on the algorithms, processes might also need sufficient storage room for storing temporary results when not enough registers are available. In the current implementation, 8 longwords are reserved for each FIFO queue accessed on each microengine (this is the `port` struct, described on page 49). Therefore the mapper is limited in the amount of local memory available to use for FIFO queues.

Next-neighbour Rings

The next-neighbour registers on each microengine can be used in three modes: an extra set of general purpose registers, one FIFO channel with a size of 128 longwords between neighbouring microengines, or as 128 separate registers readable by the neighbouring microengine. When using the registers as one FIFO queue, atomic `get()` and `put()` operations are provided, but it might be a waste of space as there are only 7 next-neighbour channels available and not all channels need to be 128 elements long. Instead, the space of 128 longwords can be divided into multiple FIFO channels, with self-implemented head and tail pointers.

Either way, the mapping of the processes to threads is crucial when selecting FIFO channels that should be mapped onto these next-neighbour channels. In such a mapping source nodes are placed on the lower microengine numbers, and sink nodes on the higher microengine numbers, as most communication flows from the source nodes to the sink nodes. Next-neighbour rings are illustrated in Figure 4.3 as number 2.

Reflector Rings

Each microengine has 128 SRAM read registers and 128 DRAM read registers. It is possible to let one microengine write data to the read registers of another microengine. This is called a reflector operation. This provides a similar solution to using the next-neighbour register banks as multiple self-implemented FIFO queues. The difference is that the next-neighbour registers have dedicated data paths to the microengines, whereas reflector operations must use the standard bus between all the elements of the network processor. Therefore it is slower. The data flow of reflector operations is shown as number 3 in Figure 4.3.

The DRAM read register banks of the microengines used for mapping processes onto, are not used in any way and can be completely used for this purpose. SRAM read registers

are used for communication with all other FIFO implementations, so these should be used more economical for this purpose.

Hardware Assisted Scratch Rings

On the scratchpad unit 16 sets of special registers (rings) are available, serving as FIFO head and tail pointers for memory allocated onto scratchpad memory. The minimum size of such a FIFO is 128 longwords, which is in most cases more than enough. When using all 16 scratchpad rings with the minimum size, half of the scratchpad memory is thereby allocated. On all 16 rings atomical `get()` and `put()` are provided. For the first 12 rings, a hardware assisted test for either fullness or emptiness is provided, but not both simultaneously. We only make use of the emptiness tests. The scratchpad rings are illustrated as number 4 in Figure 4.3.

A requirement of hardware assisted scratch ring data memory, is that its allocation needs to be aligned to the size of the ring. This means that a ring of size 128 can be allocated at longwords 0 – 127, 128 – 255, etc. A ring of size 1024 can only be assigned to longwords 0 – 1023, 1024 – 2047, etc. All hardware assisted scratch rings are sorted by the mapper on descending size, and mapped to the beginning of scratch memory in that order. This way, all rings will be correctly aligned according to their ring sizes.

Self Implemented Scratch Rings

When not all scratchpad memory is used by the hardware assisted scratch rings, the remaining part can be used by self implemented rings, assigning head and tail pointers. This is illustrated as number 5 in Figure 4.3. This implementation is also used for communication of sink nodes with the microengine responsible for transmitting data to the MSF, as an extra pointer is implemented to provide synchronisation. This synchronisation is needed when one run of the process network has completed. The transmitting microengine needs to know which value on the ring is the last value for that that run, such that it can wrap up and transmit a network packet. This process is described in section 6.5

SRAM Rings

SRAM rings are the third hardware supported FIFO implementation. Each SRAM memory channel has a queue descriptor table which can hold 64 values. As the IXP2400 has two SRAM memory channels, a total of 128 rings is readily available. The values of the queue descriptor tables can be cached into SRAM memory. This way, queue descriptors can be swapped between the queue descriptor table and SRAM memory, allowing for a number of SRAM rings only limited by the amount of available SRAM memory. The caching of

queue descriptors is however a costly operation, so it should be avoided. SRAM rings are shown in Figure 4.3 as number 6.

DRAM Rings

Theoretically it would be possible to also implement rings on DRAM memory. It is however so slow that this is not feasible. It is better to use more SRAM rings instead.

4.2.3 Strategies

Despite the many mapping options for FIFO queues, the basic rule is to map the most frequently used FIFO channels on the fastest hardware options, if the minimum size of the FIFO channel permits such a mapping. There is a trade-off between mapping a few large frequently used FIFO channels on fast memory, or putting many small less frequently used FIFO channels onto the same memory. At the moment, COMPAAN does not provide information about size or frequency of use, so optimizing this trade-off can not be done automatically yet.

Instead, our mapper assumes all FIFO channels are equal in terms of frequency of use. Mapping is done based on first-come-first-serve, where hardware assisted scratch rings are used first, and after that the self implemented scratch rings.

When next neighbour rings are implemented, the mapping is mostly a matter of the process mapping strategy. The FIFO channel mapper still should assign these rings when the processes are mapped for this purpose.

Using local memory rings is of course restricted to self-loop FIFO channels and FIFO channels between processes that are mapped onto the same microengine.

Memory Allocation

A common issue for using scratch memory and sram memory, is preventing the compiler to use the portion of memory assigned to ring data for other variables. For each memory type, the compiler can be informed to only use a portion of that memory type, starting at a specified memory address. When all ring data is assigned as one contiguous block of memory starting at address 0, and the compiler is provided with the first address following the ring data block, this issue is solved.

For scratch memory, it is important that first all hardware assisted rings are mapped in descending size order, and the self implemented scratch rings thereafter. This ensures correct alignment of the hardware assisted scratch rings.

4.3 Implementation

The IMCA tool as described in section 1.2 is implemented using the Java programming language. The mapping stage requires three input elements: the KPN specification, a set of mapping strategies and a platform specification.

The KPN is specified in an XML formatted file with information about the nodes and FIFO channels. This specification is parsed and converted into a set of Java objects, implemented by the classes `ixppn.datamodel.kpn.*`. Example classes are `Link`, `Entity` (for Nodes), `WritePort` and `ReadPort`.

The platform is also specified as a set of objects, which are implemented by the classes `ixppn.datamodel.platform.*`. An UML diagram of these classes is provided in Figure 4.4 on page 47. All method and property details are omitted in this figure. The platform object structure is generated using default constructor values, but will be made configurable using an XML specification as well.

The class `ixppn.operations.Kpn2Ixp.java` performs the mapping process. The strategies are implemented as member functions of this class. One example is the `Links2FifosGreedy()` function. It takes all `Link` objects, and assigns them to available objects extending the `FIFO` class, such as `ScratchRingHW`. Other member functions implement more fine-grained mapping tasks. The `Entity2Thread()` function is used when a node is to be mapped on a thread. This function will copy all necessary information from the node object into the thread object. It is best practice to copy all information instead of just linking to the mapped object. This way the code generation step will not be dependant on the KPN class specification.

The next step, converting the platform objects into microengine C, is described in the next chapter.

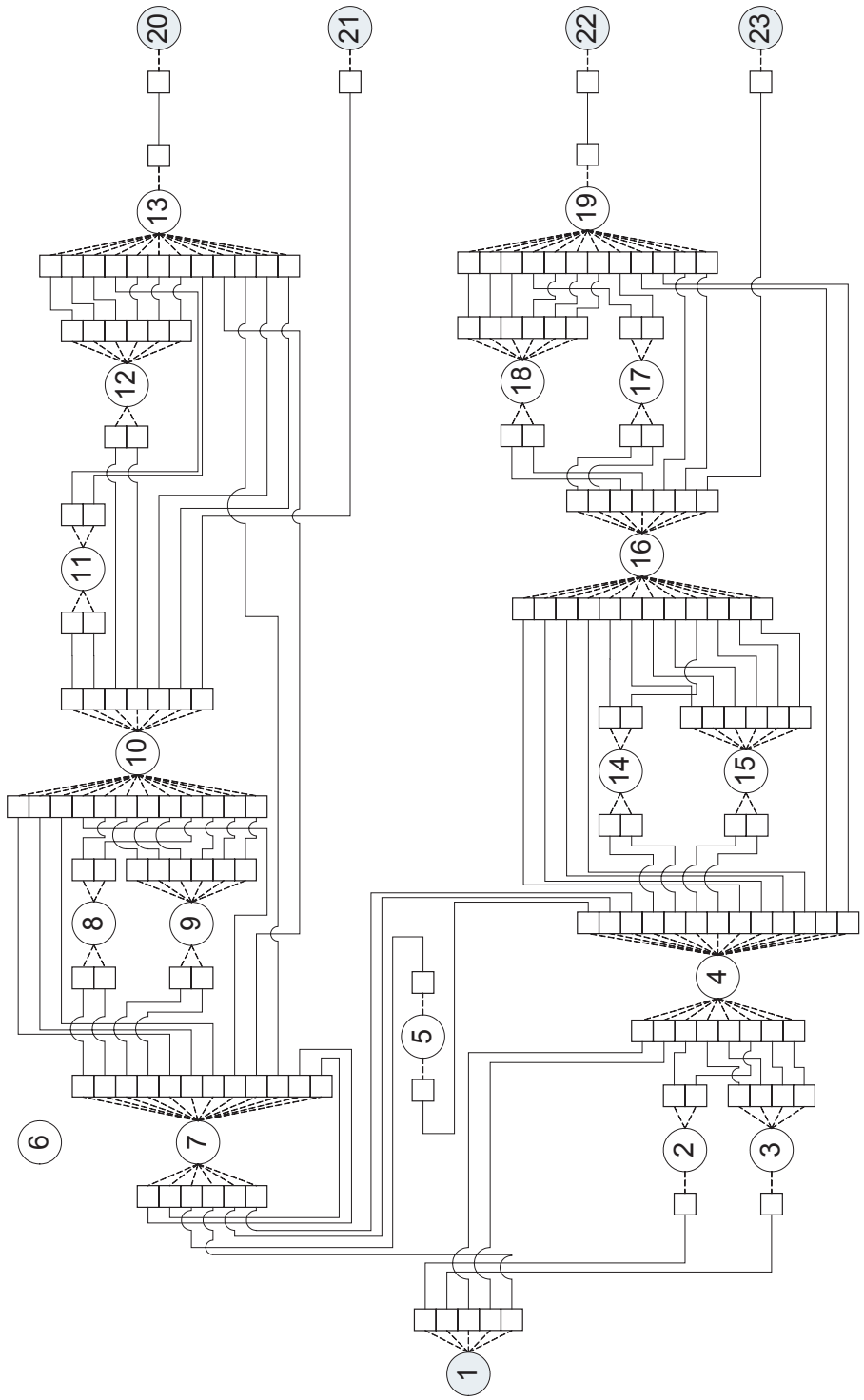

Figure 4.1: Schema

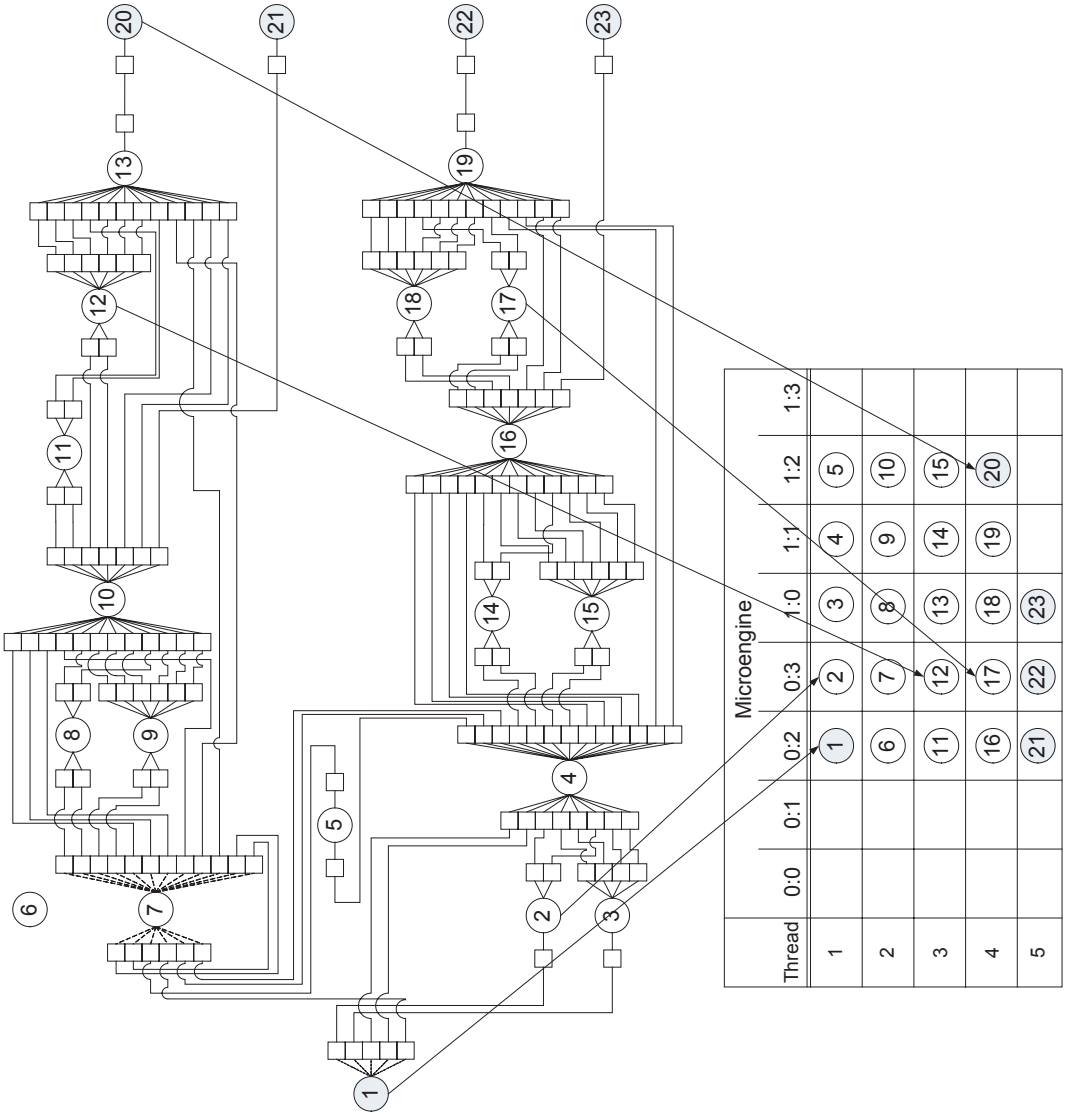

Figure 4.2: Schema with mapping

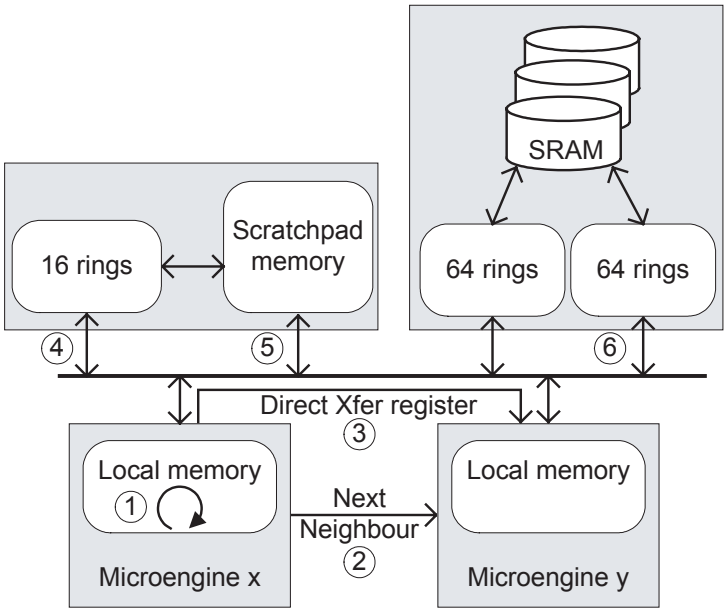

Figure 4.3: FIFO options

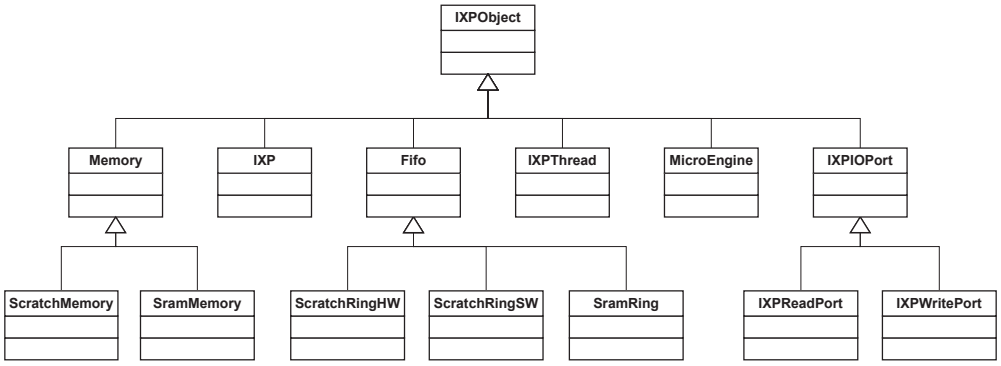

Figure 4.4: Platform Class Structure

Chapter 5

Code Generation

When all processes and FIFO channels are mapped onto hardware resources, code has to be generated for each hardware element, using the available information of the processes and FIFO channels. All operations on the IXP2400 hardware are directed by the microengines (and by the XScale core we don't use). Thus, all mapping information has to be expressed in microengine instructions.

5.1 FIFO code

For implementing the various FIFO options, the IMCA mapper tool provides a library with a common interface for all available FIFO implementations. Having this uniform model, all FIFO queues are equal with regard to how processes interact with the FIFOs. Code generation for the processes is therefore independent of the chosen mappings for the FIFO queues.

The interface consists of two layers. The first layer (`ports.h`) only includes `get()` and `put()` functions. This layer shields the process code from the FIFO queue implementations. Each FIFO implementation has its own set of functions, together forming the second layer. Such functions include tests for emptiness and fullness, as well as the actual `get()` and `put()` functions. These FIFO specific functions are operated by the first layer. For example, a value will only be placed onto a ring when it is first asserted that the ring is not full.

Communication between the process code and the first layer is done through a C struct called *port*. The port struct contains information about which FIFO implementation is used, the ring number (in case of hardware assisted scratch rings or SRAM rings), the base, head and tail pointer locations, the size and the amount of room that is available. Not all struct members are needed for all implementation types. Each microengine that is to use a certain FIFO queue will be provided with the same struct information. The port struct is needed as an argument to the `get()` and `put()` functions to specify the

FIFO. The first layer will perform a switch operation on the FIFO type specified in the port struct. From there the necessary steps, specific to the implementation type, are taken to perform the requested operation.

The port structs also act as a local cache of the state of a FIFO, using the *room* variable. If a process writes to a queue and the first layer determines there is room for another 10 tokens, it will not need to check the queues status until another 10 tokens are written as it there is only one process writing to that queue. Similarly, when a process that reads from a queue and the first layer finds out there are 10 more tokens waiting, it can read those before checking the fullness of the queue again. This saves access time.

For self implemented FIFO options (currently only the scratchpad memory version), the `port` struct also caches the head pointer for processes reading from a FIFO and the tail pointer for processes reading from a FIFOThis way, the process does not have to read the pointer from memory, as it can predict the contents by using the cache; the process is the only process of altering that specific pointer value.

5.2 Process code

Each process in the KPN specification consists of a sequential application that runs concurrently to the other processes. The sequential application is specified as a syntax tree for each node. This syntax tree is an deterministic description of the operations that have to be performed inside the process, and the order in which they are executed. A syntax tree is easily converted into any programming language specification.

The elements of the syntax tree are `for` statement nodes, `if` statement nodes and `code` statement nodes. An example of a syntax tree is given in Figure 5.1. It is the syntax tree of `ND_3` of the QRvr algorithm KPN (see Figure 2.4 on page 17). A conversion of this syntax tree to C code is given in Figure 5.2 on page 53. We use this C code conversion to explain the semantics of the syntax tree.

The `for` loops iterate over the *domain* of the process. The outer loop iterates over domain parameter `k` (lines 03 – 30) and one inner loop iterates over domain parameter `j` (lines 04 – 29). The loops correspond to `for` statement nodes of the syntax tree. Each iteration of the inner loop processes one element of the domain.

Each domain element uses the same function call to the function `Vectorize()` (line 18). The function has two input parameters and two output parameters. The two input values are obtained from two of the input FIFO channels. The two output values are written to two of the output FIFO channels.

Which input FIFO channels and which output FIFO channels to use, is dependant on the current domain element. The selection of these channels is done by `if` statements (lines 05, 08, 11, 14, 20, 23 and 26) based on the domain parameters. Inside the `if` statement, input variables are popped from the FIFO channel using the `GetPort()` function and output variables are pushed onto the FIFO channels using the `PutPort()` function.

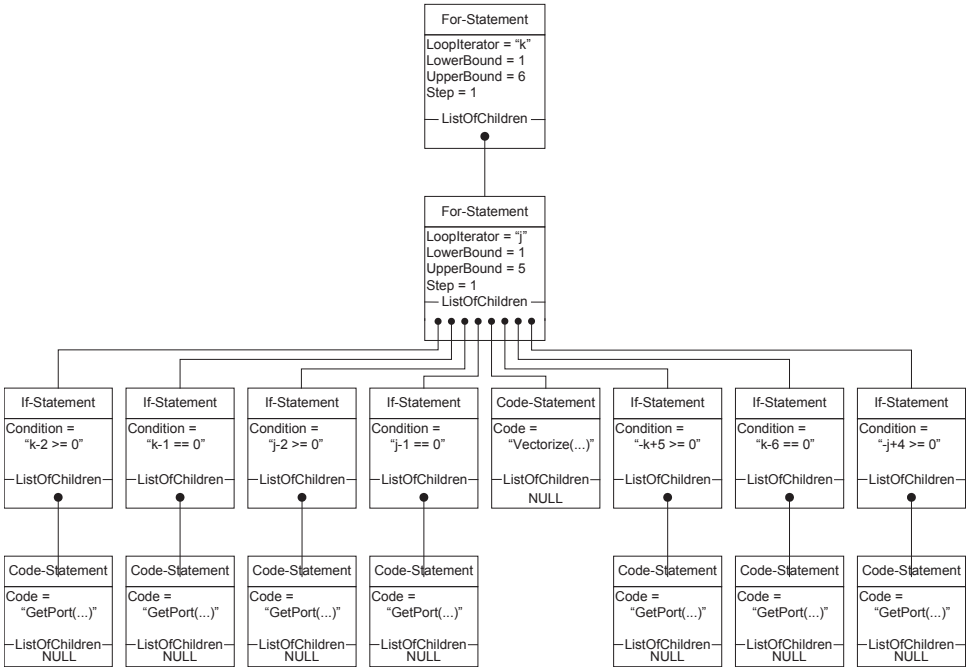

Figure 5.1: Syntax Tree

5.2.1 Microengine Code Example

Each microengine may have multiple process mapped onto its threads. For each process, program code like that of Figure 5.2 has to be generated, running it its own thread. An example of microengine C code with multiple processes running on the separate threads is provided in Appendix A. Also shown are functions and port initializations.

Lines 1 – 5 provide necessary files to include. Lines 7 – 9 are signals used for synchronizing all microengines after all FIFO structures are initialized. The signalling process is shown in lines 121 – 131. It is similar to the code shown in Figure 3.6 at the left hand side, on page 29. At line 122, a signal is sent to a central microengine. All microengines running mapped processes send the signal `init_ready_sig` to the central microengine, after all FIFO structures are initialized. When the central microengine has received the `init_ready_sig` signal from all microengines running mapped processes, it sends the signal `ring_ready_sig` back to the first thread of these microengines. This signal is caught at line 125. Now, thread 0 sends the signal `thread_0_ring_ready_sig` to the next thread at line 126. Thread 1 wakes up at line 129, and signals the next thread at line 130. Subsequent threads use the same mechanism as thread 1.

Lines 11 – 24 declare all port structs. Lines 32 – 119 initialize the port structs. Inside the `main()` function, each thread has private instruction code, separated by `if`

statements with `ctx()` conditions which branch on the current thread number (0 – 7).

In the current mapping situation any application will run once. By placing an extra `while` loop around the `for` loops for each process, the application will run continuously. This was done to obtain results for chapter 7.

5.3 Implementation

For conversion of KPN specifications into microengine C, IMCA makes use of the Visitor [20] design pattern implemented in Java. This design pattern is a way of separating an algorithm from an object structure. A Visitor class is an interface that that has a `visit()` method for each class of the object structure. Each class of the object structure has an `accept()` function, which simply calls back to the `visit()` function of the calling visitor. A benefit of this system is that new operations can be added to a structure by adding a new Visistor class, without modifying the structure itself. Another benefit is that the result of a computation can depend on the runtime types of its arguments. For example, when specific code needs to be generated for each FIFO channel, it suffices to call `Fifo.accept(theVisitor)` for each FIFO. The exact runtime type of the FIFO will determine whether `visit(ScratchRingHW)`, `visit(ScratchRingSW)` or `visit(SramRing)` will be run. Each function will provide some functionality specific to the object type it operates on. In the case of code generation, each function will output the code used to implement the object, based on the properties of the object.

An overview of the Visitor classes used for IMCA is provided in Figure 5.3 on page 54. The abstract IXPPlatformVisitor class describes the traversion options for the platform objects. The IXPCodeGenVisitor extends IXPPlatformVisitor, and implements the code generation for each platform object, as obtained from the mapping step of section 4.3.

The result of using the code generator visitor, is a microengine C file for each microengine, like the file provided in Appendix A. These files, together with a set of fixed library functions, provide the input to the Intel C Compiler for Network Processors.

We provide a brief explanation of the working of the code generator visitor. The starting point is an IXP platform object, and a `CodeGenVisitor` object. The `accept()` function is called on the platform object, with the visitor as parameter. The `accept()` function returns a call to the `visitStructure(IXP)` function. This function invokes `accept()` on each microengine. A call to `visitStructure(MicroEngine)` for each microengine is the result. This function creates a new file for the current microengine, and produce lines of code like `main() {` and other lines that only occur once for the microengine. For each thread, `accept()` is invoked. This results in the function `visitStructure(IXPThread)` producing code for each thread (lines 136 – 165, etc). The initialization code of each port (lines 32 – 36, etc) is created by invoking `accept()` on each FIFO, resulting in a `visitStructure()` call to either `ScratchRingHW`, to `ScratchRingSW` or to `SramRing`, each generating the initialization code for that FIFO.

```
01  main{} {
02    int k, j, in_1, in_0, out_2, out_0;
03    for (k=1 ; k <= 6 ; k += 1) {
04      for (j=1 ; j <= 5 ; j += 1) {
05        if (k-2 >= 0) {
06          GetPort(&ND_3IP_1, &in_0);
07        }
08        if (k-1 == 0) {
09          GetPort(&ND_3IP_2, &in_0);
10        }
11        if (j-2 >= 0) {
12          GetPort(&ND_3IP_3, &in_1);
13        }
14        if (j-1 == 0) {
15          GetPort(&ND_3IP_4, &in_1);
16        }
17
18        Vectorize(&out_2,&out_0,&in_1,&in_0);
19
20        if (-k+5 >= 0) {
21          PutPort(&ND_3OP_1, &out_0);
22        }
23        if (k-6 == 0) {
24          PutPort(&ND_3OP_1_d1, &out_0);
25        }
26        if (-j+4 >= 0) {
27          PutPort(&ND_3OP_3, &out_2);
28        }
29      }
30    }
31  }
```

Figure 5.2: Process C Code

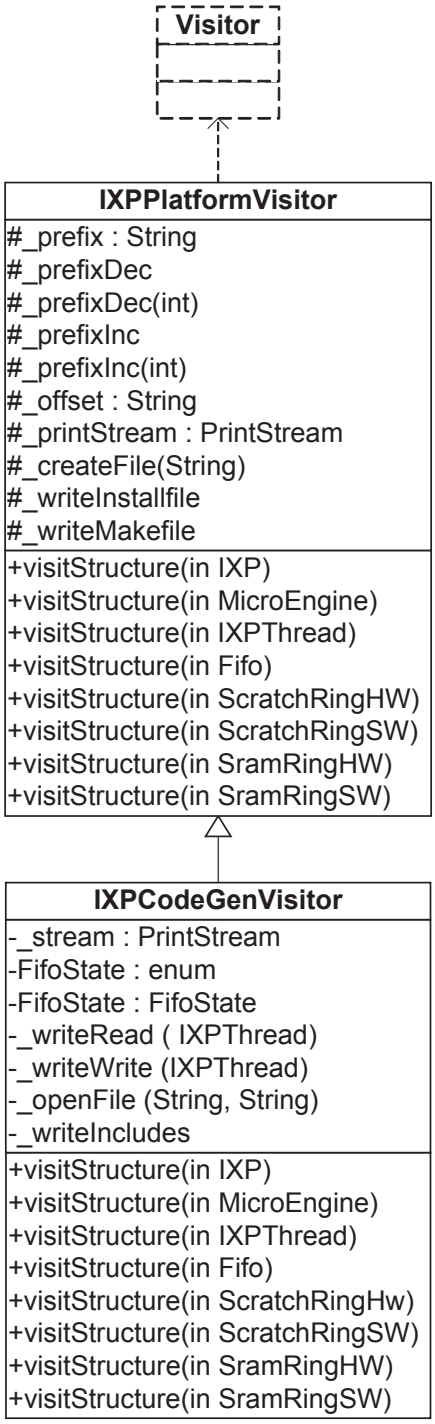

Figure 5.3: UML of Visitor

Chapter 6

Receive and Transmit

Not explicitly denoted by a KPN specification is the need for interaction of the source and sink nodes with the outside world. To guarantee a fast throughput of data, all threads of three microengines are used by IMCA. One microengine is used to transport incoming network packets from the MSF device into DRAM memory. The second microengine reads the payload information from these packets and places it onto FIFO queues to be read by source node processes. It also reads from FIFO queues coming from the sink node processes and creates new packets in DRAM memory using this data as payload information. The third microengine is used for transmitting these newly assembled packets from DRAM memory to the MSF device where it will be send out again. This type of operation is typical for Network Processor applications. A consequence for using three microengines is that only five will be available for mapping application processes.

In this section the process of receiving, reading, assembling and transmitting packets is described. The whole flow, with its mapping onto the microengines, is represented in Figure 6.1 on page 56. In this figure is represented how data is being transferred by all eight microengine (ME) 0:0 from the MSF onto the DRAM RBUF. The first four threads of microengine 0:1 will then transfer the data onto the first FIFO which will be read out by the first node planted on one of the treads on microengine 0:2 - 1:2 and represents the sourcenode of the KPN of the application. The data will be processed through the KPN. At the right of the KPN the last thread which represents the sinknode, will put the data onto a FIFO which will be read by the last four threads of microengine 0:1. The data will be put onto DRAM TBUF. All eight threads of microengine 1:3 will get the data from the TBUF and put it onto the MSF.

6.1 Receiving packets

Network packets arrive at the MSF interface of the IXP2400 as described in section 3.4. These network packets contain the data which forms the input of the application which

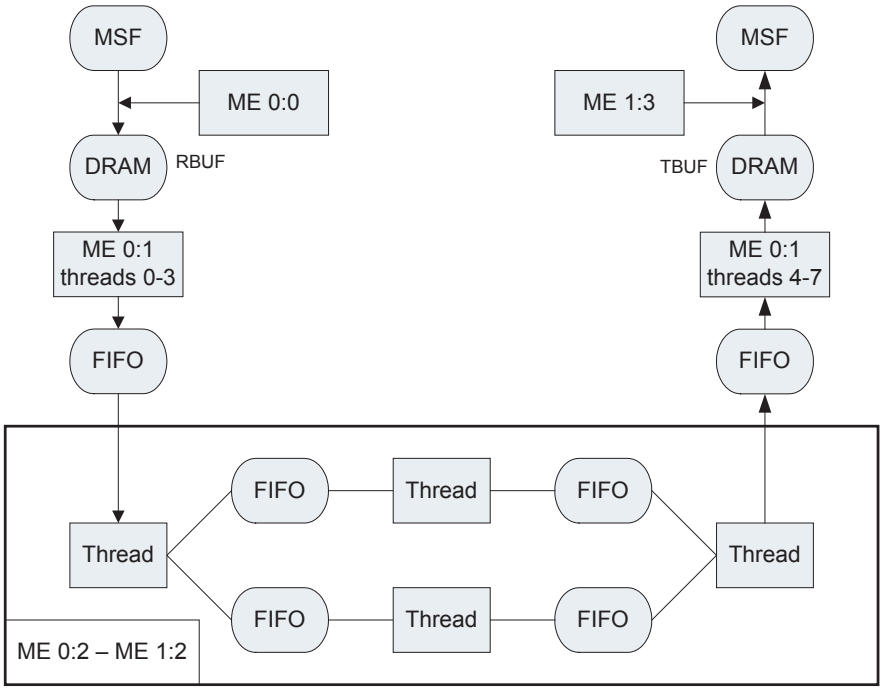

Figure 6.1: Flow

will be executed. To make the network packet data available, the complete network packets are copied and stored into DRAM. For this copy process we make use of a standard mechanism which is also being used by most of the network applications for the IXP. There is a direct physical link between MSF and DRAM, so data doesn't have to go through the microengines. The microengines control the copying process. This process of receiving and copying the data on the MSF device and from there onto DRAM is done by eight cooperating threads of one single microengine, therefore this process is quick and smooth. As seen in Figure 6.1, this process is handled by microengine 0:0. Because this process of receiving and copying the data works independent of the application which uses the data and runs on a different microengine, this mechanism is very flexible. Taking these advantages into account, we decided to adapt the existing code and use it for our purposes.

As described in section 3.4, network packets are divided into small partial packets, called mpackets. The receivebuffer (RBUF) of the MSF is in fact an array of mpackets. These mpackets will be copied into DRAM in the correct order, thus making the original network packets available to the next processing step. The result is a ringbuffer of packets in DRAM of 32 MB in size in total. For every packet there is an equal size reserved of 2048 byte which includes 8 byte for a header. In this header the length

of the packet is written. This results in a simple and structured array with fixed size elements, irrespective whether the element is full or not, rather than allowing variable sizes to be written into DRAM.

All eight threads of the microengine do in principle exactly the same work, independent from one another. Each thread will processes an mpacket in a consecutive way, one thread after another. This makes it possible that eight mpackets are being processed concurrently. In Figure 6.2 is shown how each thread works.

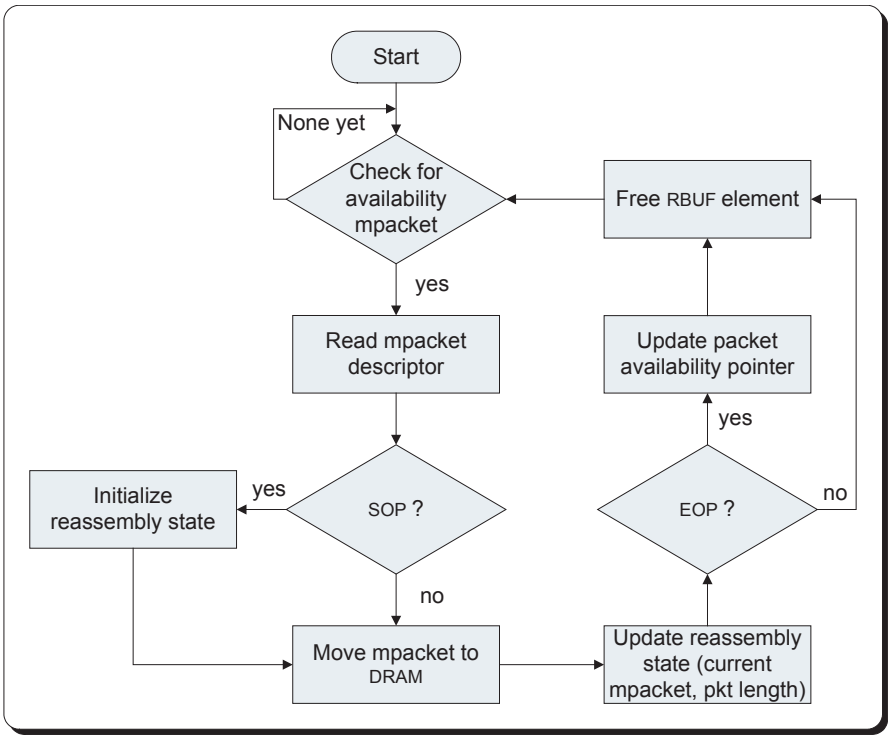

Figure 6.2: Flowchart of a packet reassembly from mpackets

Figure 6.2 is an abstraction of the process because error situations are not included. As soon as the process has started, it will check for availability of mpackets, for as long as there are none available. When one mpacket becomes available, the descriptor of the mpacket will be read. The descriptor can be one of the following:

- SOP: This mpacket is the start of a new packet. A new reassembly state in a new buffer element has to be initialized and the data can be moved there.
- EOP: This mpackets is the end of the packet.
- Both EOP and SOP, this means that this single mpacket is the whole packet.

- ERR: An error has occurred with this mpacket; if needed, the assigned buffer element will be freed.
- *empty*: This mpacket is neither the start, nor the end of the packet, but an mpacket somewhere in-between the start and end of the packet.

Updating the reassembly state means updating the pointers and counters to the new mpackets and the packet length. After an EOP occurrence the general packet availability pointer can be updated and this will mean that the application can read the data of the packet from the RBUF.

This mechanism can run for ever. For now, our whole application only runs one time and for many applications all input data for one run fits in one packet with a maximum size of 2 KB. This means only one thread will copy one mpacket. As soon as our mapping is suitable for running applications repeatedly, this process does not need to be adapted; at it endlessly handles subsequent packets already.

The data throughput speed obtained by the implementation discussed in this section, is provided in section 6.4.

6.2 Reading and Assembling Packets

Now that the packets are aligned in DRAM, the contents of the payload of the packets have to be imported to the implemented KPN network sequentially. The output of the sink-nodes will have to be partitioned into sendable packets. This functionality has also been implemented by us.

In order not to use too many resources, these two functions have been mapped onto one microengine. This will save us one microengine and specific data which is necessary for both processors (such as the header information of the packets) can be stored locally. The two functions of reading and writing the packets will each use 4 threads of one microengine.

6.2.1 Reading Packets

The algorithm of section 6.1 uses a variable in SRAM (`fast_wi`, fast write index) to specify which packets in the packetbuffer are ready to be used. By reading this variable the program knows which packets from DRAM can be used. In the current implementation there is a restriction that no more than one source-node can accept input data, however it is possible to have multiple source-nodes which generate data themselves. In order to allow for multiple source nodes accepting input data, the packet reading process would need to discriminate between packets. For example, each source node can have a private IP address, which needs to be read by the packet reading process. Based on that information, the packet data will be sent to the correct source node.

As soon as a packet is ready in DRAM, the payload needs to be read. This begins on integer number 54 (an integer is 32 bit, each of 4 byte) after the header. With the packetlength stored in the header, the program can deduct how many integers need to be read. The memorywidth of DRAM is 4 integers, so per memory read instruction there are directly 4 integers available. Figure 6.3 illustrates how the data will be put on a FIFO by making use of the `put()` code described in section 5.1. This FIFO will act as the source where the source-nodes can get their data from.

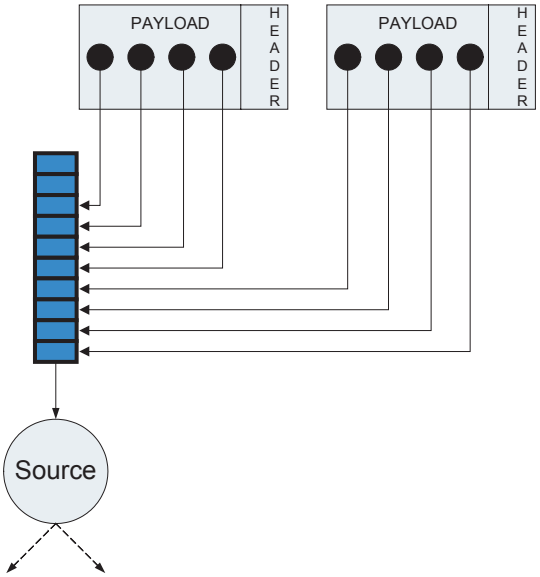

Figure 6.3: Receiving on one sourcenode

As soon as the fist packet is being read, a copy of the header is saved into local memory. This copy will serve as a template to make new packets.

6.2.2 Creating Packets

As soon as results arrive from the sink-nodes, they will be stored in packets which will be send out. These packets are stored in DRAM. The format in which they are stored is the same as that of the packets which are being received, with the same fixed size packet length, but they are stored in their own buffer: TBUF. The current implementation has the restriction that there is only one sink-node from which data can be send.

The begin of each new packet will be filled with the header as it was stored in local memory. Attributes like the MAC- and IP-address will be changed and are fixed. For future work we can advice to make these configurable (see chapter 8). The packet will then be filled with data from the FIFO originating from the sink-node as illustrated in Figure 6.4 on page 60. A packet is ready to be send out as soon as either the maximum

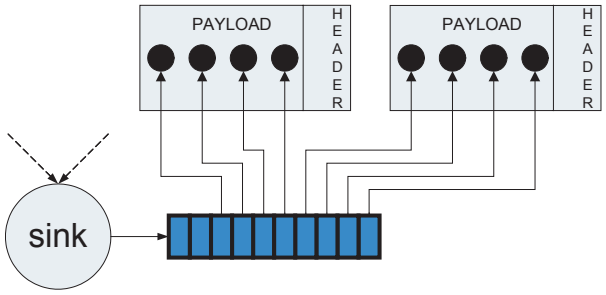

Figure 6.4: Sending one mpacket

packet size is reached (by TBUF) or as soon as the last value of a run of the application has been taken from the FIFO. This last action (the synchronization between sink and sending) is being described in section 6.5. The final and correct packetlength in the header will be corrected later. At this moment an index variable in SRAM (`global_counter`) is being increased, to inform the sending microengine that the next packet is ready to be send out.

6.3 Transmitting Packets

As soon as the packets are assembled in DRAM, they are ready to be transmitted out on the network. This is done via the MSF. The data packets in DRAM need to be partitioned into mpackets to meet the MSF requirements. The first mpacket of the assembled datapacket will get a SOP-flag, and subsequently the last mpacket will get a EOP-flag. As soon as all mpackets of one packet have been placed onto the MSF, they will be transmitted onto the network. A visualization of the process of transmitting the packets is shown in Figure 6.5 on page 63.

6.4 Speed

The Intel IXP2400 can receive and transmit data at a speed of 2.5 Gbps. The implementation of the receive and transmit processes we use, operates roughly at a speed of 1 Gbps. All received data has to be put on one FIFO queue (section 6.2.1), one integer of 32 bit at a time. In ideal circumstances (only one thread active, immediate availability of data, using hardware assisted scratch rings) it takes about 150 clock cycles to put one integer on the FIFO queue. Running at a speed of 600 MHz, this results in a throughput of 128 Mbps. Compared to the current receive speed of 1 Gbps, this FIFO queue is roughly 8 times slower. Compared to the maximum receive speed of 2.5 Gbps, it is roughly 20 times slower. Speeds similar to 150 clock cycles are obtained for reading data from

a FIFO queue. We assume the throughput speed of the whole KPN implementation is at most as slow as one FIFO queue.

The current speed available by the receiving and transmitting processes is an overkill, as the current FIFO implementations form a bottleneck. A solution to is to speed op FIFO implementations. Chapter 8 suggests a solution, where per FIFO access multiple integers can be written. Another option is to slow down the receiving and transmitting processes, by using less than eight threads per process. For example, the receive and transmit processes could be mapped to run on 4 threads of a microengine each. This will result in saving one microengine which can be used for mapping KPN processes onto.

6.5 Stop Signals

As soon as the sink node has completed outputting all data for one run of the application, the packet generating process needs to know what data element on the FIFO is the last from that run. This information cannot be transferred using the existing FIFO synchronisation of blocking read and blocking write. Therefore, we created an extra status pointer in the software implemented scratch rings. This stop-pointer refers to the last data element in the FIFO belonging to the latest completed run (it is only used between the sink node and the packet creating process). Furthermore, the sink process will synchronize with the packet generating process using signals, to inform the packet generating process that the stop-pointer is about to be set. The process is described as follows:

1. The tread performing the sink node functionality detects that the last token for a run of the application is about to be put on the FIFO channel. The stop pointer is set equal to the tail pointer value. The thread sends a signal to the packet generating microengine, and waits for a signal in return.

2. The packet generator receives the signal, and sends a signal in return. This is similar to the process described in Figure 3.6. From now on, it will check for every read operation on the FIFO channel whether the stop pointer is equal to the head pointer.

3. The sink receives the return signal, and finally puts the last token on the FIFO.

4. If there were more tokens on the FIFO, these will be consumed by the packet generator by putting them in the packet currently being built, as their FIFO positions are not equal to the stop pointer.

5. The head pointer is equal to the stop pointer. The packet generator will consume this last token too, placing it in the packet. The packet will be finished up and the

transmitting microengine will be informed that it is ready to be sent out. A new packet will be prepared in TBUF.

This procedure ensures that the packet generating process will not try to read the final token of a run, before it knows it is the final token. Without this type of synchronisation, it cannot be ensured that too many tokens (i.e. the first tokens from the next run of the application) are put in the result packet of the last completed run. If there are no subsequent runs, without this synchronization it cannot not be ensured that the packet generating process will not deadlock, trying to read the next token while having consumed the last token already.

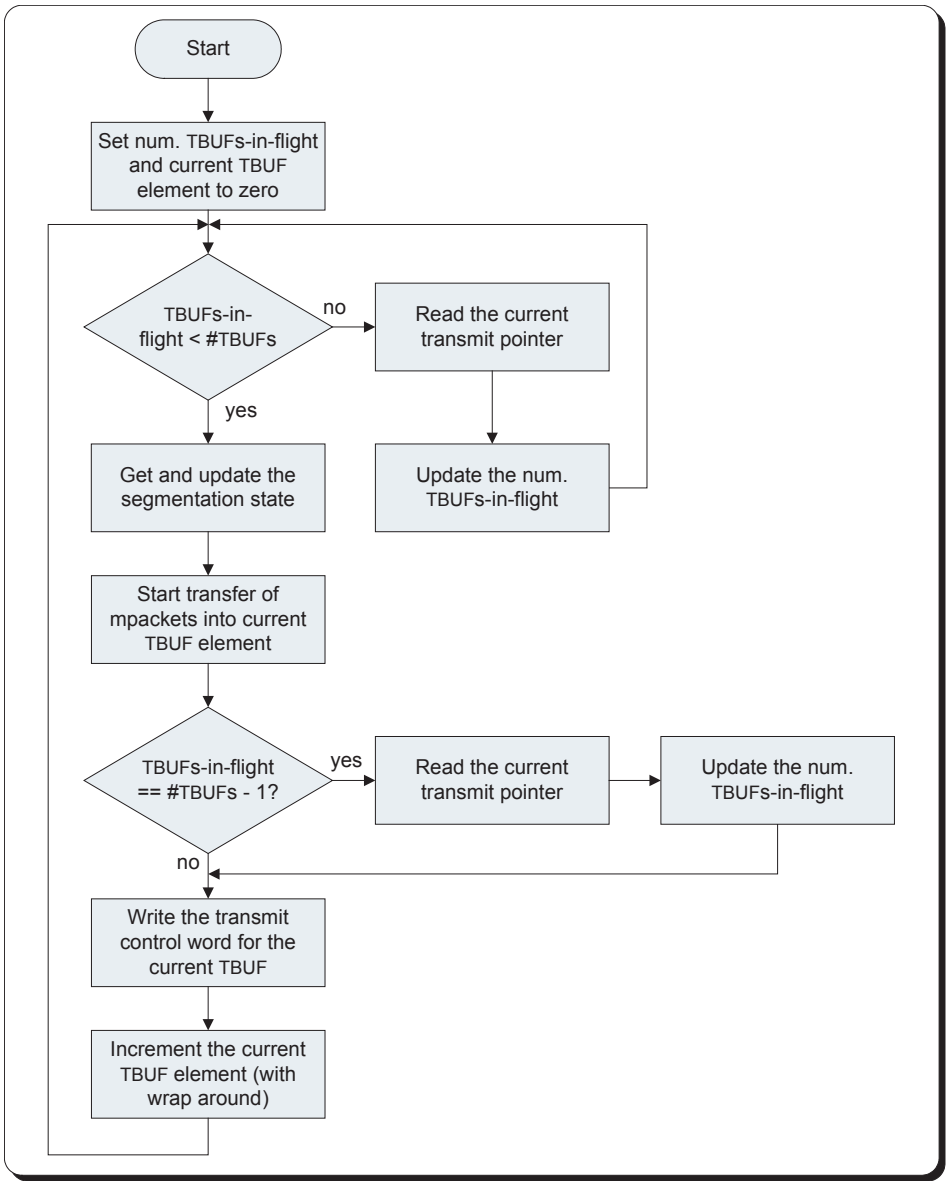

Figure 6.5: Flowchart assembling an mpacket into a packet to be transmitted

Chapter 7

Experiments and Results

In this chapter we present performance measurements from implementations obtained via the IMCA tool. We measure two aspects of the performance: the *throughput* of inbound data of the implementation expressed in megabits per second (Mpbs) and the computing power expressed in millions operations per second (MOPS). These values heavily depend on the algorithm which is used. We used two algorithms, QRvr and FDWT. Both are discussed in this chapter.

The goal for this thesis is to investigate whether the mapping of a KPN specification onto the IXP hardware can be performed automatically, not to provide a better performance than existing solutions. Therefore, we do not compare our results with other implementations. Instead, we provide these figures to show that our solution is valid and to provide an affirmative answer for the research question. We have provided an initial implementation of the IMCA tool, thus these are initial performance results; additional research efforts will undoubtedly provide improved performance. Some suggestions for improving performance are provided in chapter 8.

We had to overcome many difficulties for implementing all work presented in this thesis. All elements had to work correctly together to be able to conduct performance measurements. Therefore, only at a late stage of our research efforts we were able to obtain the results we present in this chapter.

7.1 Method

In this section, we explain which methods we use to obtain results expressed in Mbps and MOPS from our measurements.

Per process, we measure the amount of clockcycles it takes to complete one run of the application. The first clockcycle is when the first `for`-loop is entered, and the last clockcycle is when the process exits the last `for`-loop. Given that the hardware runs at a clockfrequency of 600 Mhz, the amount of time spent in each process is calculated.

The amount of operations performed per run per process is the amount of operations needed for each assignment statement inside the process, times the frequency in which the assignment statement has been used. Division of the amount of operations by the amount of time spent in a process, results in the amount of operations per second per process (MOPS). Adding up the MOPS information of all processes together, as all processes run in parallel, results in the MOPS information for the whole implementation.

As all processes start together for the first run, the duration of the process that takes the longest time to complete, is the duration of one run for the whole implementation. When the application is run several times consecutively, again the duration of the process that takes the longest time to complete, is the duration of the whole implementation. This process is the bottleneck for the application.

For each application the size of the input for each source node per run is known. Combined with the duration of one run of the application, the amount of data that is processed per second (Mbps) is computed.

7.2 QRvr

The QRvr algorithm is a relatively simple algorithm. The KPN version produced by COMPAAN, contains only 5 nodes and 12 FIFO channels. Each node is mapped onto a different microengine (as there are 5 available for mapping) and all FIFO channels are mapped onto fast hardware assisted scratchpad memory rings. Needing such few resources, we were able to execute an implementation in a relatively early stage of the development of the IMCA tool. The downside for using this application, is that we do not have a full implementation of the algorithm to our avail. For these experiments, the functions `Vectorize()` and `Rotate()` thus perform only dummy operations.

A figure of the KPN specification is provided in Figure 2.4 on page 17. For this discussion, it is of interest that both source nodes (1 and 2) perform simple operations of providing data and only execute as often as there are tokens in the input data, whereas nodes 3 and 4 execute more often as they repeatedly process each others data.

Node	Clockcycles	Seconds	Operations	MOPS
1	1671	0,0028	15	5,39
2	3262	0,0054	30	5,52
3	40183	0,0670	30	0,45
4	39912	0,0665	60	0,90
5	40247	0,0671	15	0,22
Total				12,48

Table 7.1: Performance results QRvr, 1 run

Table 7.1 shows the results for a single run of the QRvr implementation provided

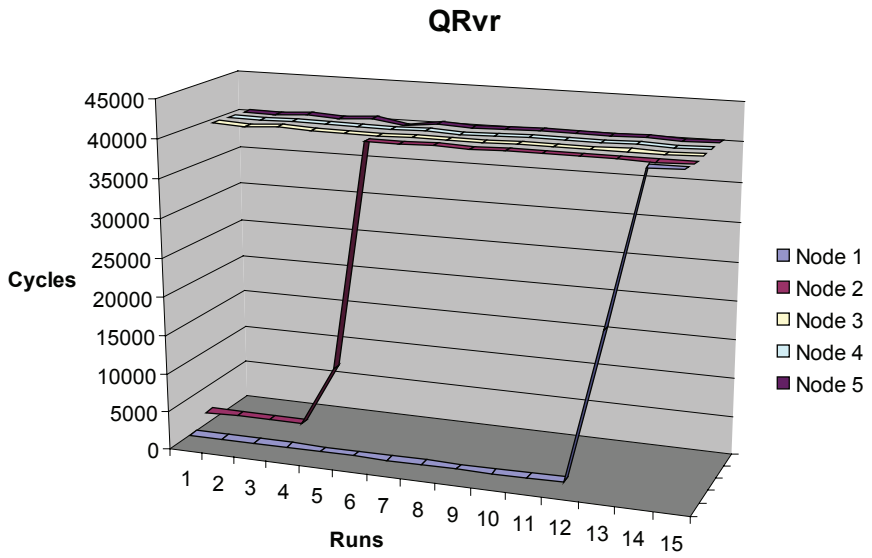

Figure 7.1: Conformation of processes to bottleneck speed in QRvr

by the IMCA tool. Nodes 1 and 2 are finished much more quickly than the other nodes, resulting in a high performance statistic for these nodes, as they spend only a small amount of time on processing all their data. However, this does not provide a realistic figure for two reasons. First of all, we use dummy functions which only perform one operation. Secondly, in a realistic usage the application runs continuously. The effect of running the algorithm multiple times, is shown in Figure 7.1. The horizontal axis shows the number of runs performed consecutively. The vertical axis represents the amount of clockcycles each node takes to complete for that run. For each process node, a graph is shown. At first, nodes 1 and 2 are quickly finished, while the other nodes take more time (about 40.000 cycles). After a few runs, the FIFO channels between the source nodes and the other nodes become full, as the reading processes are slower than the writing processes. Now, nodes 1 and 2 have to wait for enough room to become available on the FIFO, before they can put new data onto it. The result is that with the chosen FIFO channel sizes, all processes become as slow as the bottleneck process after about 15 runs. Therefore, if clockcycle values of 40.000 are used in Table 7.1 for nodes 1 and 2, the total performance results in a value of 2 MOPS.

However, the figure results of 12.5 MOPS and 2 MOPS are generated with dummy functions which only count 1 operation per function call. A fair estimation for a real implementation is that both functions `Vectorize()` and `Rotate()` each perform 20 operations per function call. In that case, a single run of QRvr results in a total performance of 38 MOPS, and when ran multiple times still 28 MOPS.

Frame	Frame size	Bits	Frames/s	Mbps
Input	15	480	14908	7.15
Output	30	960	14908	14.31

Table 7.2: Speed results QRvr

Table 7.2 shows the throughput results for the QRvr algorithm. In our experiments, the highest amount of clockcycles we measured for a process, was 41247. Supposed that each run takes 41247 cycles to complete, the algorithm can run 14908 times a second; or the algorithm processes 14908 *frames* per second. Each input frame consists of 15 tokens, and each output frame consists of 30 tokens for the QRvr implementation we used. Now, if each token is represented as an integer of 32 bits, a throughput of 7.15 Mpbs is achieved for incoming data, and 14.31 Mbps for outgoing data.

7.3 FDWT

A more defying mapping effort is that of the FDWT algorithm. The KPN version of FDWT consists of 23 nodes and 41 FIFO channels. Therefore, a successful mapping has been much more troublesome to achieve than for the QRvr algorithm. The combined memory space required for all FIFO channels to guarantee deadlock-free operation exceeds the amount of scratchpad memory available on the IXP2400. A successful FIFO channel mapping was dependent on our SRAM ring implementation, which took long to become bug-free. Therefore, we are proud of being able to perform an operational mapping for the FDWT algorithm after all. Moreover, we have a full implementation for this algorithm, such that the result figures are more representative that that of the QRvr algorithm. The only aspect of the FDWT algorithm that is not implemented by our solution, is that the algorithm has 4 sink nodes, whereas our implementation only supports actual output for one sink node. This issue is described in chapter 8.

Runs	MOPS	Frames/s	Input Mbps
1	25	1043	34.18
> 1	26	1048	35.55

Table 7.3: Results FDWT

The results of the FDWT algorithm are shown in Table 7.3. Since there are no less than 23 nodes, we only present the final results for the whole implementation, and not per process. The particular implementation of FDWT we used processes images with a size of 32 by 32 pixels; thus each input frame consists of 1024 tokens.

In contrast to QRvr, the FDWT algorithm *increases* speed after the first run. The difference is that the bottleneck of QRvr is found in back-end nodes of the KPN, such that

the front-end nodes finish quickly as long as there is plenty room in the FIFO channels. In contrast, with FDWT the bottleneck resides in the front-end nodes. FDWT is an image processing algorithm, where two pixel rows must be buffered completely into FIFO channels before other nodes perform operations on vertically neighbouring pixel values. In the first run, the whole network remains idle before two pixel rows are buffered. In consecutive runs, new pixel rows are buffered while the rest of the network is processing previous pixel rows. Therefore, the back-end of the KPN network then does no longer have to wait for the buffering of pixel rows, as it has already been done during previous runs. The speedup is only small, as after the first run 26 MOPS are achieved compared to 25 MOPS for the first run. The number of input frames processed per second changes from 1043 to 1048, which results in a throughput on the input node changing from 34.18 to 35.55 Mbps.

Chapter 8

Improvement Suggestions

The process of creating methodologies like we created for the IMCA tool, is rather time consuming when starting from scratch. This is largely due to the fact that mastering the programming skills required for IXP platform requires a lot of time and practice. Even after months, mistakes are still easily made. Implementing all ideas we present in this thesis, turned out to be impossible for the amount of time available. Our implementation works; but not really optimal yet. Therefore, we provide an overview of issues we find worth improving. Some have been mentioned already, some have not. For those who continue research based on our results, we strongly encourage to take these suggestions as a starting point for further implementation efforts.

- Using local caches for FIFO channel data. This has multiple reasons. First, a larger class of KPN applications can be used, as there exist variations that need to read a token multiple times, or need to read a token at a certain position in the FIFO queue. Secondly, the whole data flow can be speeded up by a significant factor. Memory operations on the IXP allow for multiple integers to be read or written simultaneously, up to 8 without speed penalty or up to 16 with a slight speed penalty. If there are 8 integers waiting on the queue, these can be read at once, eliminating the need for 7 other slow memory operations. Of course, these values need to be cached into local memory. The `port` struct can be adapted to do so. As for *writing* data to a FIFO channel, a process might choose to buffer outbound tokens locally, before sending multiple tokens at once. This requires carefulness, for not to cause deadlocks. A solution to prevent deadlocks is flushing this outbound buffer when a time restriction is violated, i.e. after a certain amount of time in which no new tokens are put in the buffer.

  This buffering technique is especially useful for buffers that need to process large amounts of data. The packet reading process would benefit enormously, as the network packet payload can be put onto the FIFO in much less costly memory operations. The sink nodes would also benefit from buffering outbound tokens,

as the packet generation process will certainly not cause deadlocks and only be happy to accept larger amounts of tokens at once. There are no negative side effects on buffering inbound data, except for the use of more local memory resources. When also using local buffers for outbound data, in combination with automatic flushing at timeouts, there is a negative side effect on FIFO channels that never can exceed a certain size smaller than the buffer size. These FIFO channels will only be filled when a timeout occurs. The smaller the maximum channel usage size, the worse the problem is. Sadly, many FIFO channels never exceed a size of 1. Setting the timeout value not too high reduces the problem. The trade-off between causing extra delays for not forwarding available data the next process is waiting on, and the elimination of slow memory operations by reading/writing multiple values at once, is the key question in this matter.

- Using the XScale core. At least one process could be mapped onto the XScale core. It may not seem much to be able to map one extra process, but it doesn't need to share processing time with other threads. Therefore a process with a high workload can be mapped onto the XScale core. There are probably ways of running multiple processes on the XScale too.

- Implementing next-neighbour FIFOs. The hardware is there, and our code is flexible in terms of adding new FIFO options.

- Implementing local memory FIFOs. This would save a lot of time for FIFO channels with both ends on the same microengine. This is only an option for FIFO channels with a small minimum size. In practice most FIFO channels have a minimum size of 1.

- Creating mapping strategies to make optimal use of the next-neighbour and local memory FIFO channels.

- Making use of the Strategy design pattern, to allow for different mapping strategies more easily.

- Making attributes configurable, such as the IP address and MAC address to which packets are to be sent.

- Making the platform specification configurable, using XML formatted files.

- Adding tools in the design flow that calculate the minimum FIFO sizes for the network not to deadlock.

- Tuning the receive, transmit and packet reading/creating processes. In the current implementation three microengines are used for these processes together. It is

probably possible to use less resources without introducing a bottleneck for the streaming data flow.

- Allowing for more source nodes which provide data from outside. Each source node should be addressed by some property in the network packets, or some header in the payload of the packet. Based on this information, the packet reading function could decide on what FIFO the contents of the packet should be put. This is illustrated in Figure 8.1.

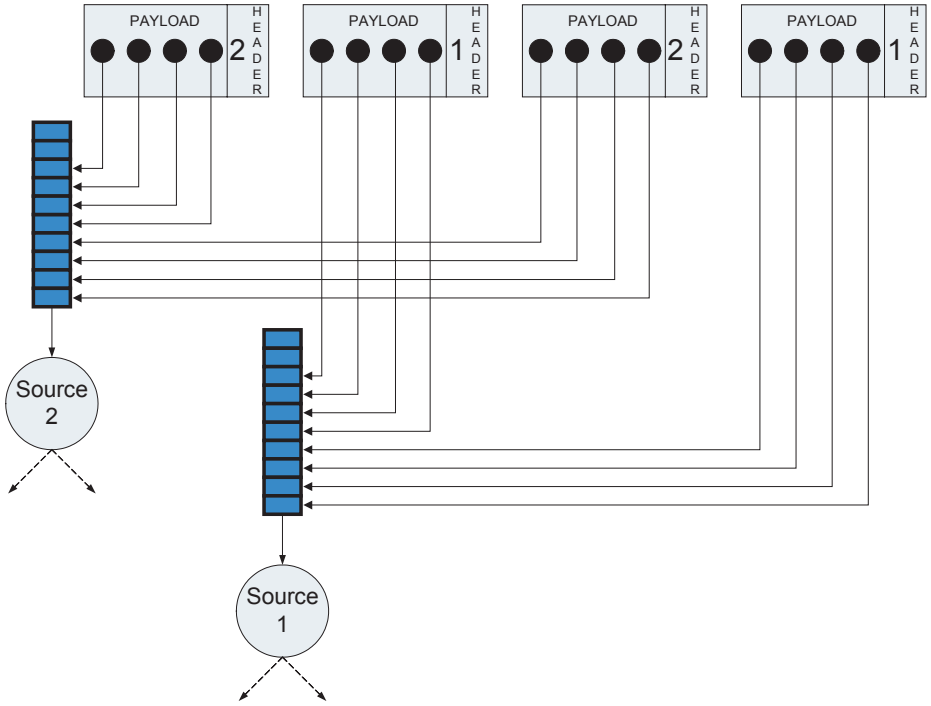

Figure 8.1: Receiving on two sourcenodes

- Allowing for more sink nodes to transmit data. The packet generating threads should assemble multiple packets together in TBUF, one for each sink node. A simple solution is to assign one packet generating thread to each sink node. The amount of allowed sink nodes is then equal to the amount of packet generating threads.

- Real automatization of the design flow. At the moment, the mapper generates a set of `.c` files, which need to be inserted manually inside a standard project using the Developer Workbench. In this standard project, most configuration is already correct, such that configuration `.list` files do not need to be created. Instead, all files should be created by the IMCA tool, and the compilation should be performed using the command line version of the Intel C compiler for Network Processors.

Chapter 9

Conclusions

The world is becoming multiprocessor oriented, as it is the only option to match modern performance requirements. Multiprocessor systems are becoming increasingly complex, with an increasing amount of transistors. Programming efficiency also increases, but at a lower rate than the increase of the platform complexity. Therefore, the so called *productivity gap* is growing. Programmers need more efficient tools to match their productivity to a level which is required by the hardware design.

An example of multiprocessor systems are the Intel IXP Network Processors. Our hypothesis is that these network processors are an ideal platform for execution of stream-based applications like digital signal processing and image processing. However, these network processors are hard to program and require a steep learning curve. Moreover, programming multiple tasks running concurrently is a hard task for the human mind. This prevents high productivity for this platform and is in essence an example of a productivity gap. If the hypothesis is true, low productivity rates would still prevent many applications being implemented on the hardware, despite the hardware benefits.

To circumvent the need for complicated programming, a tool is required that allows for simple application specification for applications targeted at the IXP platform. Our research question is whether it is possible to perform an automated mapping of a simple application specification onto the platform of the Intel IXP2400 Network Processor. The possibility of such an automated mapping allows developers to make use of the IXP2400 network processor for streaming applications with a short development time.

We have shown that the combination of Compaan and imca exactly provide the tools needed for an automated mapping of an easily sequentially specified application onto the multiprocessor IXP hardware. The developer writes algorithms in Matlab as static affine nested loop programs, which have a single line of control and are thus much more easy to specify than concurrent specifications of the same algorithm. The Compaan compiler transforms this specification into an equivalent kpn specification. The imca tool transforms the kpn specification into an implementation which is executable on the IXP2400 platform. The process of receiving input data from an ethernet connection,

and the process of transmitting output data onto the ethernet connection, are also implemented. Therefore, we conclude that our research question is answered affirmatively.

With regard to the performance results, we think that the actual use of Intel IXP Network Processor hardware is at least an interesting option for running streaming applications, especially as there is room for performance speedups by making more use of the hardware capabilities. We feel that the performance results are encouraging to perform further research, which should exploit the optimizations proposed in chapter 8 and perform comparisons with implementations on other hardware.

Appendix A

Example Microengine C File

```
001 #include "ixp.h"
002 #include "util.h"
003 #include "scratch_rings.h"
004 #include "ports.h"
005 #include "functions.c"
006
007 __declspec(visible) SIGNAL ring_ready_sig;
008 __declspec(remote)  SIGNAL init_ready_sig3;
009 SIGNAL thread_0_ring_ready_sig;
010
011 __declspec (shared local_mem aligned(4)) port_type ND_3IP_2;
012 __declspec (shared local_mem aligned(4)) port_type ND_3OP_1;
013 __declspec (shared local_mem aligned(4)) port_type ND_8IP_12;
014 __declspec (shared local_mem aligned(4)) port_type ND_8OP_1;
015 __declspec (shared local_mem aligned(4)) port_type ND_13IP_20;
016 __declspec (shared local_mem aligned(4)) port_type ND_13IP_21;
017 __declspec (shared local_mem aligned(4)) port_type ND_13IP_22;
018 __declspec (shared local_mem aligned(4)) port_type ND_13IP_23;
019 __declspec (shared local_mem aligned(4)) port_type ND_13IP_24;
020 __declspec (shared local_mem aligned(4)) port_type ND_13OP_2;
021 __declspec (shared local_mem aligned(4)) port_type ND_18IP_32;
022 __declspec (shared local_mem aligned(4)) port_type ND_18OP_1;
023 __declspec (shared local_mem aligned(4)) port_type ND_18OP_1_d1;
024 __declspec (shared local_mem aligned(4)) port_type ND_23IP_41;
025
026 void ports_init() {
027   __assign_relative_register((void *)&thread_0_ring_ready_sig, 7);
028   if (ctx() == 0) {
029     __assign_relative_register((void *)&ring_ready_sig, 6);
030     __assign_relative_register((void *)&init_ready_sig3, 3);
031
032     ND_3IP_2.type              = SRAM_RING;
033     ND_3IP_2.ring_nr           = 60;
034     ND_3IP_2.size              = 1024;
035     ND_3IP_2.room              = 1024;
036     ND_3IP_2.base              = 12288;
037
038     ND_3OP_1.type              = SRAM_RING;
039     ND_3OP_1.ring_nr           = 53;
040     ND_3OP_1.size              = 1024;
041     ND_3OP_1.room              = 1024;
042     ND_3OP_1.base              = 40960;
043     InitPort(&ND_3OP_1);
```

```
044
045     ND_8IP_12.type              = SRAM_RING;
046     ND_8IP_12.ring_nr           = 52;
047     ND_8IP_12.size              = 1024;
048     ND_8IP_12.room              = 1024;
049     ND_8IP_12.base              = 45056;
050
051     ND_8OP_1.type              = SRAM_RING;
052     ND_8OP_1.ring_nr           = 51;
053     ND_8OP_1.size              = 1024;
054     ND_8OP_1.room              = 1024;
055     ND_8OP_1.base              = 49152;
056     InitPort(&ND_8OP_1);
057
058     ND_13IP_20.type              = SRAM_RING;
059     ND_13IP_20.ring_nr           = 48;
060     ND_13IP_20.size              = 1024;
061     ND_13IP_20.room              = 1024;
062     ND_13IP_20.base              = 61440;
063
064     ND_13IP_21.type              = SRAM_RING;
065     ND_13IP_21.ring_nr           = 47;
066     ND_13IP_21.size              = 1024;
067     ND_13IP_21.room              = 1024;
068     ND_13IP_21.base              = 65536;
069
070     ND_13IP_22.type              = SRAM_RING;
071     ND_13IP_22.ring_nr           = 46;
072     ND_13IP_22.size              = 1024;
073     ND_13IP_22.room              = 1024;
074     ND_13IP_22.base              = 69632;
075
076     ND_13IP_23.type              = SCRATCH_RING_HW;
077     ND_13IP_23.ring_nr           = 4;
078     ND_13IP_23.size              = 128;
079     ND_13IP_23.room              = 128;
080     ND_13IP_23.base              = 11776;
081
082     ND_13IP_24.type              = SRAM_RING;
083     ND_13IP_24.ring_nr           = 57;
084     ND_13IP_24.size              = 1024;
085     ND_13IP_24.room              = 1024;
086     ND_13IP_24.base              = 24576;
087
088     ND_13OP_2.type              = SCRATCH_RING_HW;
089     ND_13OP_2.ring_nr           = 15;
090     ND_13OP_2.size              = 512;
091     ND_13OP_2.room              = 512;
092     ND_13OP_2.base              = 0;
093     InitPort(&ND_13OP_2);
094
095     ND_18IP_32.type              = SRAM_RING;
096     ND_18IP_32.ring_nr           = 56;
097     ND_18IP_32.size              = 1024;
098     ND_18IP_32.room              = 1024;
099     ND_18IP_32.base              = 28672;
100
101     ND_18OP_1.type              = SRAM_RING;
102     ND_18OP_1.ring_nr           = 41;
103     ND_18OP_1.size              = 1024;
104     ND_18OP_1.room              = 1024;
105     ND_18OP_1.base              = 90112;
```

```
106     InitPort(&ND_18OP_1);
107
108     ND_18OP_1_d1.type              = SRAM_RING;
109     ND_18OP_1_d1.ring_nr           = 40;
110     ND_18OP_1_d1.size              = 1024;
111     ND_18OP_1_d1.room              = 1024;
112     ND_18OP_1_d1.base              = 94208;
113     InitPort(&ND_18OP_1_d1);
114
115     ND_23IP_41.type              = SCRATCH_RING_HW;
116     ND_23IP_41.ring_nr           = 12;
117     ND_23IP_41.size              = 512;
118     ND_23IP_41.room              = 512;
119     ND_23IP_41.base              = 6144;
120
121     // Signal filter that initialization is done
122     cap_fast_write((0<<4) | (__signal_number(& init_ready_sig3, 0x01)) |
                           (0x01 <<7), csr_interthread_sig);
123
124     // Wait for filter that all initializations are done
125     wait_for_all(&ring_ready_sig);
126     signal_same_ME_next_ctx(__signal_number(&thread_0_ring_ready_sig));
127     __implicit_read(&thread_0_ring_ready_sig);
128   } else {
129     wait_for_all(&thread_0_ring_ready_sig);
130     signal_same_ME_next_ctx(__signal_number(&thread_0_ring_ready_sig));
131   }
132 }
133
134 main() {
135   ports_init();
136   if (ctx() == 0 ) {
137     __declspec(gp_reg) int i;
138     __declspec(gp_reg) int j;
139     __declspec(gp_reg) int in_0;
140     __declspec(gp_reg) int out_0;
141     for (i=0 ; i <= 14 ; i += 1) {
142       for (j=0 ; j <= 31 ; j += 1) {
143         in_0 = GetPort(&ND_3IP_2);
144
145         copy(&out_0,in_0);
146
147         PutPort(&ND_3OP_1, &out_0);
148       }
149     }
150   }
151   if (ctx() == 1 ) {
152     __declspec(gp_reg) int i;
153     __declspec(gp_reg) int j;
154     __declspec(gp_reg) int in_0;
155     __declspec(gp_reg) int out_0;
156     for (i=0 ; i <= 15 ; i += 1) {
157       for (j=15 ; j <= 15 ; j += 1) {
158         in_0 = GetPort(&ND_8IP_12);
159
160         copy(&out_0,in_0);
161
162         PutPort(&ND_8OP_1, &out_0);
163       }
164     }
165   }
166   if (ctx() == 2 ) {
```

```
167     __declspec(gp_reg) int i;
168     __declspec(gp_reg) int j;
169     __declspec(gp_reg) int in_1;
170     __declspec(gp_reg) int in_2;
171     __declspec(gp_reg) int in_0;
172     __declspec(gp_reg) int out_1;
173     __declspec(gp_reg) int out_0;
174     for (i=0 ; i <= 15 ; i += 1) {
175       for (j=0 ; j <= 15 ; j += 1) {
176         if (j-15 == 0)
177           in_0 = GetPort(&ND_13IP_20);
178         if (j-1 >= 0)
179           if (-j+14 >= 0)
180             in_0 = GetPort(&ND_13IP_21);
181         if (j == 0)
182           in_0 = GetPort(&ND_13IP_22);
183         in_1 = GetPort(&ND_13IP_23);
184         in_2 = GetPort(&ND_13IP_24);
185
186         my_low_flt_hor(&out_0,&out_1,in_0,in_1,in_2);
187
188         PutPort(&ND_13OP_2, &out_1);
189       }
190     }
191   }
192   if (ctx() == 3 ) {
193     __declspec(gp_reg) int i;
194     __declspec(gp_reg) int j;
195     __declspec(gp_reg) int in_0;
196     __declspec(gp_reg) int out_0;
197     for (i=0 ; i <= 15 ; i += 1) {
198       for (j=1 ; j <= 15 ; j += 1) {
199         in_0 = GetPort(&ND_18IP_32);
200
201         copy(&out_0,in_0);
202
203         if (j-15 == 0)
204           PutPort(&ND_18OP_1, &out_0);
205         if (-j+14 >= 0)
206           PutPort(&ND_18OP_1_d1, &out_0);
207       }
208     }
209   }
210   if (ctx() == 4 ) {
211     __declspec(gp_reg) int i;
212     __declspec(gp_reg) int j;
213     __declspec(gp_reg) int in_0;
214     __declspec(gp_reg) int out_0;
215     for (i=0 ; i <= 15 ; i += 1) {
216       for (j=0 ; j <= 15 ; j += 1) {
217         in_0 = GetPort(&ND_23IP_41);
218
219         sink(&out_0,in_0);
220       }
221     }
222   }
223 }
```

Bibliography

[1] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettre. COMPAAN: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures.
http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/codes.pdf

[2] The MathWorks
http://www.mathworks.com/products/matlab

[3] Claudiu Zissulescu, Todor Stafanov, Bart Kienhuis, Ed Deprettre. LAURA: Leiden Architecture Research and Exploration Tool.
http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/fpl03.pdf

[4] Kai Huang and Ji Gu. ESPAM: and Application Mapping for Multiprocessor Systems On-Chip.
http://www.liacs.nl/~stefanov/pdf/KaiJi_MSc05.pdf

[5] Intel
http://www.intel.com/design/network/products/npfamily/index.htm?iid=ncdcnav2+proc_netproc&

[6] Xilinx
http://www.xilinx.com

[7] Intel IXA: Intel® Internet Exchange Archtitecture.
http://www.intel.com/design/network/papers/intelixa.pdf

[8] FFPF/Streamline.
http://www.fastnetworkprocessing.org

[9] Niraj Shah, William Plishkre, Kurt Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In: second Workshop on Network Processors (NP-2), ninth International Symposium on High Performance Computer Architectures (HPCA), February 2003.

[10] NST: Network Speed Technologies.
http://www.network-speed.com

[11] Lal George, Matthias Blume. Taming the IXP Network Processor. In *Conference on Programming Language Design and Implementation, Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, San Diego, California, USA, 2003*. ISSN:0362-1340

[12] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[13] Tom Parks. Bounded Scheduling of Process Networks. PhD thesis, University of California at Berkeley, 1995.

[14] LERC: Leiden Embedded Research Centre.
`http://www.liacs.nl/~cserc/lerc/group/sprojects.html`

[15] LIACS: Leiden Institute of Advanced Computer Science. Part of the Faculty of Mathematics and Natural Sciences of Leiden University.
`http://www.liacs.nl`
`http://www.fwn.leidenuniv.nl/english/`
`http://www.leidenuniv.nl/`

[16] Intel, *Intel IXP2400 Network Processor Hardware Reference Manual*, October 2004.

[17] ARM: Advanced Risc Machines Ltd.
`http://www.arm.com`

[18] The Microblaze soft processor.
`http://www.xilinx.com`
`http://en.wikipedia.org/wiki/MicroBlaze`

[19] Todor Stefanov, Bart Kienhuis, Ed Deprettere. Algorithmic Transformation Technique for Efficient Exploration of Alternative Application Instances. In *Proc. 10th Int. Symposium on Hardware/Software Codesign (CODES'02), Estes Park, Colorado, USA, May 6-8, 2002*.

[20] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison-Wesley, 1995.