# Synthesis of a parallel data stream processor from data flow process networks

Zissulescu-Ianculescu, C.

## Citation

Zissulescu-Ianculescu, C. (2008, November 13). *Synthesis of a parallel data stream processor from data flow process networks*. Retrieved from https://hdl.handle.net/1887/13262

Version: Corrected Publisher's Version

License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden

Downloaded from: https://hdl.handle.net/1887/13262

**Note:** To cite this publication please use the final published version (if applicable).

# Synthesis of a Parallel Data Stream Processor from Data Flow Process Networks

Claudiu Zissulescu-Ianculescu

# Synthesis of a Parallel Data Stream Processor from Data Flow Process Networks

**Proefschrift**

ter verkrijging van de graad van Doctor aan de Universiteit Leiden, op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden, volgens besluit van het College voor Promoties te verdedigen op donderdag 13 November 2008 klokke 16:15 uur

door

Claudiu Zissulescu-Ianculescu
geboren te Bucureşti, România
in 1976

Samenstelling promotiecommissie:

| | | |
|---|---|---|
| promotor | Prof. dr. Ed Deprettere | |
| co-promotor | Dr. A.C.J. Kienhuis | |
| referent | Dr. Steven Derrien | INRIA, France |
| overige leden | Prof. dr. Harry Wijshoff | |
| | Prof. dr. Joost Kok | |
| | Prof. dr. Kees Goossens | Technische Universiteit Delft, NXP Eindhoven |
| | Dr. Laurens Bierens | Eonic BV, Delft |

Synthesis of a Parallel Data Stream Processor from Data Flow Process Networks
Claudiu Zissulescu-Ianculescu. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN/EAN 978-90-9023643-8

Copyright ©2008 by Claudiu Zissulescu-Ianculescu, Leiden, The Netherlands.
All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission from the author.

Printed in the Netherlands

*soţiei mele Dani*

# Contents

**1 Introduction 1**

1.1 COMPAAN Data Flow Process Network . . . 3
1.2 Problem Definition . . . 6
1.3 Solution Approach . . . 7
1.4 Thesis Contribution . . . 9
1.5 Related Work . . . 9
  1.5.1 Hardware Architecture Implementations that uses the Polyhedral Model 9
  1.5.2 Hardware Architecture Implementations that uses the Process Network MoC . . . 11
1.6 Thesis Outline . . . 12

**2 From COMPAAN Data Flow Process Network to Abstract Architecture 13**

2.1 Background . . . 13
2.2 Topological Mapping . . . 18
2.3 Semantic Mapping . . . 19
2.4 Conclusions . . . 23

**3 Control Synthesis 25**

3.1 Control Synthesis in Read/Write Control Units . . . 26
3.2 The Look-up Table Controller . . . 28
  3.2.1 Example . . . 29
  3.2.2 Discusion . . . 30
3.3 The Parameterized Predicate Controller . . . 30
  3.3.1 Example . . . 32
  3.3.2 Discusion . . . 33
3.4 The Partitioned Parameterized Predicate Controller . . . 34
  3.4.1 Example . . . 36
  3.4.2 Discussion . . . 36
3.5 Conclusions . . . 37

**4 Communication Synthesis 39**
4.1 Background . . . . . 39
4.1.1 The Order of Producing and Consuming Tokens . . . . . 40
4.1.2 The Lifetime of a Token . . . . . 41
4.1.3 Communication Types: Overview . . . . . 42
4.2 Communication Channel Template in Abstract Architecture . . . . . 43
4.2.1 The Channel Template for the Extended Linearization Model Realization . . . . . 44
4.2.2 The Extended Linearization Model Modifications in the Read Unit and Write Unit Controllers . . . . . 45
4.3 FPGA Realization . . . . . 46
4.3.1 Example . . . . . 46
4.4 Conclusions . . . . . 48

**5 Memory Bound Estimation 49**
5.1 Maximum Size of FIFO channels . . . . . 49
5.2 Volumes and Lexical Addressing Functions . . . . . 51
5.3 In-Order Channel Upper-bound Memory Estimation . . . . . 52
5.4 Self-loops Channel Memory Estimation . . . . . 53
5.5 Memory Estimation using Bounding Boxes . . . . . 57
5.5.1 Background . . . . . 57
5.5.2 Deriving the Bounding Boxes . . . . . 58
5.6 Conclusions . . . . . 62

**6 Expression Synthesis 63**
6.1 Related Work . . . . . 64
6.2 The Approach . . . . . 65
6.2.1 High-Level Optimizer . . . . . 65
6.2.2 Low-Level Optimizer . . . . . 66
6.3 Simplification . . . . . 66
6.3.1 Example . . . . . 68
6.4 Method of Differences (MoD) . . . . . 69
6.5 Predicated Static Single Assignment . . . . . 69
6.6 Examples of Implementations of Expressions . . . . . 73
6.6.1 Example 1 . . . . . 73
6.6.2 Example 2 . . . . . 74
6.7 Conclusions . . . . . 78

**7 IP Core Integration 81**
7.1 Embedding an IP Core . . . . . 82
7.1.1 Handling of Pipeline’s Stalls . . . . . 83
7.2 Profiler . . . . . 84
7.2.1 Increasing the Pipeline Utilization of a Virtual Processor . . . . . 87
7.2.2 Case Study . . . . . 89
7.2.3 Discussion . . . . . 92
7.3 Conclusion . . . . . 93

**8 Case Studies** **95**
- 8.1 Subspace Tracking . . . 95
- 8.2 The Matrix-Matrix Multiplication Algorithm . . . 96
  - 8.2.1 Discussion Matrix-Matrix Multiplication Implementation . . . 98
- 8.3 The QR Factorization Algorithm . . . 99
  - 8.3.1 Discussion Matrix QR Factorization Implementation . . . 100
- 8.4 The Matrix SVD Decomposition Algorithm . . . 100
  - 8.4.1 Discussion Matrix SVD Decomposition Implementation . . . 104
- 8.5 Discussion . . . 105

**9 Conclusions** **107**

**Bibliography** **109**

**Index** **117**

**Acknowledgment** **119**

**Samenvatting** **121**

**Curriculum Vitae** **123**

Chapter 1

# Introduction

An embedded system is an information processing system that is application domain specific (e.g., signal processing, multimedia, automotive, communications) and tightly coupled to its environment. Tightly coupled to the environment means that the system must react to incoming data at a speed that is imposed by the type and properties of that data. For example, a DVD player has to read, decode and display the movie (the incoming data) at a rate such that the user can observe a smooth transition between two consecutive frames. Thus, embedded systems are reactive systems that are very often real-time systems.

The computational requirements of today's embedded systems are such that a single processor can not provide the compute power. Instead, new platforms are emerging that are able to satisfy the performance needs of tomorrow's embedded applications. These new platforms are usually multi-processor or multi-core execution platforms consisting of a number of processing elements and a communication, synchronization and storage infrastructure, all integrated on a single chip. These systems are called multiprocessor system-on-chip (MPSoC). A MPSoC may be homogeneous or heterogeneous. All processing elements in a homogeneous MPSoC are of the same type, e.g., instruction set architecture (ISA) elements. On the other hand, the heterogeneous MPSoC systems are composed of processing elements that are not the same. These elements may be software programmable (ISA), hardware programmable, or even dedicated. The processing elements may operate autonomously, or may be co-processing elements. A co-processing element is a processing element that executes complex ISA instructions in a shorter period than the outsourcing ISA element could.

Moreover, multi-processor execution platforms may be *given* or may be *dedicated*. A given platform is a platform that has properties of its own (e.g., Intel processors, IBM Cell processor or graphical processor units). A dedicated platform is a platform that can be partially or totally reconfigured. Such a platform is the Field Programmable Gate Array (FPGA). The FPGA execution platforms are special in that they are not pre-defined (except for fairly general admissible organizations), but can be application customized without the programmer having to deal with the platform specification. FPGAs may consist of embedded CPUs or DSP blocks, distributed RAMs, specialized input/output blocs, and configurable logic blocs (CLBs).

To use the parallelism available in FPGA execution platforms, we need to program them in such a way that we can exploit *distributed control* and *distributed memory*. Distributed control means that the individual components on a platform can proceed autonomously in time without considering other components. Distributed memory means that data is not pooled in a large global memory, but distributed over the platform. Although distributed memory and control are key requirements to take advantage of the new emerging platforms, we observe that applications are typically cast in the form of a sequential imperative programming language, i.e., a Matlab, a C/C++ or a Java program. A strong point of the imperative model of computation is that it is easy to reason about a program as only a single thread of control needs to be considered. Also, the memory space is global, i.e., all data comes from the same memory source. However, the single memory and the single thread of control are contradictory to the need for distributed control and memory. Therefore, programming these new platforms is a very tedious, error prone, and time consuming process.

There are two ways in which we can overcome the programming problem. One way is to require application developers to specify their applications in a parallel programming language (textual or graphical). Graphical or visual programming styles have been proposed and successfully used to specify *streaming data* signal processing and multimedia applications. Typical examples of such parallel programming styles rely on *dataflow graph* and *dataflow process network* models of computation (MoC) [1, 2]. In these models, a program consists of active entities (functions, threads, processes) that communicate point-to-point over FIFO channels. Application developers are reluctant to provide specifications in terms of these models for several reasons. Firstly, the models are either not expressive enough or are undecidable. Secondly, practical applications can not be specified only in terms of dataflow models that do not really take dynamic control flow into account.

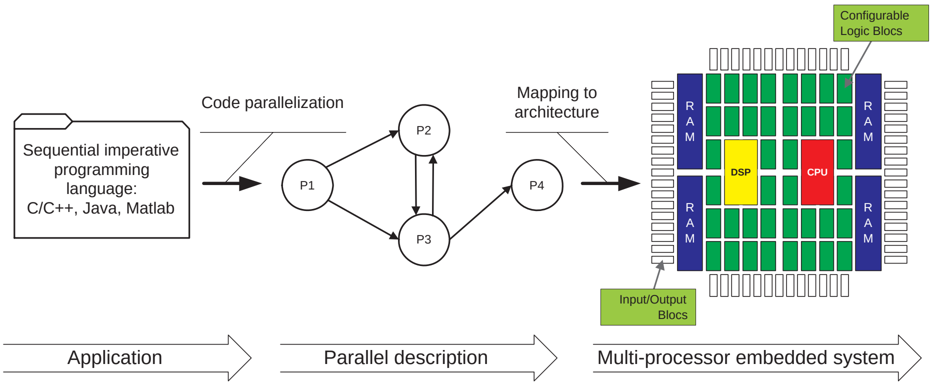


Figure 1.1: Mapping of an application to an FPGA execution platform

The other way to overcome the mismatch between a sequential imperative application specification and a targeted parallel execution platform is to convert (to parallelize) the sequential specification to an input-output equivalent parallel specification that is a better match to a targeted multi-processor execution platform. Then, the parallelized code is mapped onto the multi-processor execution platform. The action of converting the parallelized code to an executable code suited for the multi-processor execution platform is sometime called *mapping* [3] and sometime called *synthesis* [4].

In this last approach, the programmer (almost) does not need to know about parallel programming or parallel architectures in order to exploit inherited parallelism in the application. This approach is illustrated in Figure 1.1 in which the multi-processor execution platform is embedded in an FPGA execution platform. Application developers will most likely continue using sequential imperative languages to specify applications, because these languages are expressive general purpose languages. Thus, the second approach deserves further investigation.

Not every sequential imperative language program can be easily - and preferably automatically - converted to an input-output equivalent parallel specification. However, in signal processing, multimedia, molecular biology, and other related application domains, there are nested loop programs (NLP) of which many can be converted to input-output equivalent parallel specifications. In particular, those that are so called affine nested loop programs can automatically be converted. The conversion to input-output equivalent parallel specification of a subset of these nested loop programs has been amply studied and reported in the literature [5–8]. This subset of the nested loop programs is called *static affine nested loop programs*. In this thesis, I focus on the problem of mapping the input-output parallel specification of a static affine nested loop programs to FPGA multi-processor execution platforms.

More specifically, we address the problem of synthesizing *Process Network* specifications to FPGA multi-processor execution platforms. The process networks we consider are special cases of Kahn Process Networks [2]. We call them COMPAAN Data Flow Process Networks (CDFPN) because they are provided by a translator called the COMPAAN compiler that automatically translates affine nested loop programs to input-output equivalent (COMPAAN) process network specifications [5]. COMPAAN Dataflow Process Network is a model of computation that expresses an application naturally in terms of distributed control and distributed memory. The CDFPN programs are parallel programs that specify networks of active entities (threads, processes or actors) that communicate point-to-point over unbounded communication channels. The inter-process synchronization is done by means of a *blocking read* protocol. This protocol states that a process can always write to a channel, but it blocks when it attempts to read from a channel that is empty.

Our objective is to provide an effective and efficient implementation of CDFPNs in an FPGA execution platform, where our implementation is close to a one-to-one mapping of the originating CDFPN. However, in our implementation we do not make use of the embedded CPU blocks, specialized DSP blocks or soft-cores processors. The execution platform emerges as part of the mapping process resulting in a dedicated multi-processor execution platform for a given CDFPN specification.

## 1.1 COMPAAN Data Flow Process Network

COMPAAN Data Flow Process Network is a model of computation well suited to represent applications from the realm of digital signal processing. In this section, we sketch the behavior of a CDFPN that is equivalent to an imperative nested loop program.

The CDFPN MoC communication semantics is similar to the Kahn Process Network (KPN) communication semantics [2]. The KPN MoC is more general than CDFPN MoC, which is closer to the Dataflow Process Network (DPN) [1], preserving the monotonicity property [2] (i.e., a CDFPN needs only partial information of the input stream in order to produce partial information of the output stream). As in the case of the DPN, the CDFPN processes map input tokens into output tokens in concordance with a set of *firing rules*. These rules dictate precisely what tokens must be available at the input for the process to fire. A firing consumes input tokens and produces output tokens. In the CDFPN case, the firing rules are derived by the COMPAAN compiler using the *Polyhedral Model* [7, 9, 10]. The construction of the CDFPN firing rules is covered in [3]. Here, the notion of *variants* is introduced, representing a set of firing rules. A CDFPN process produces a single scalar token when it fires.

Listing 1.1: A simple Matlab program

```
for i = 1 : 1 : 1,
  [ z(i) ] = Init();
end

for i = 1 : 1: N,
  [z(i+1)] = bar(z(i));
end

for i =N+1 : 1 : N+1,
  [] = Sink(z(i));
end
```

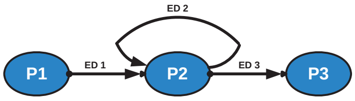
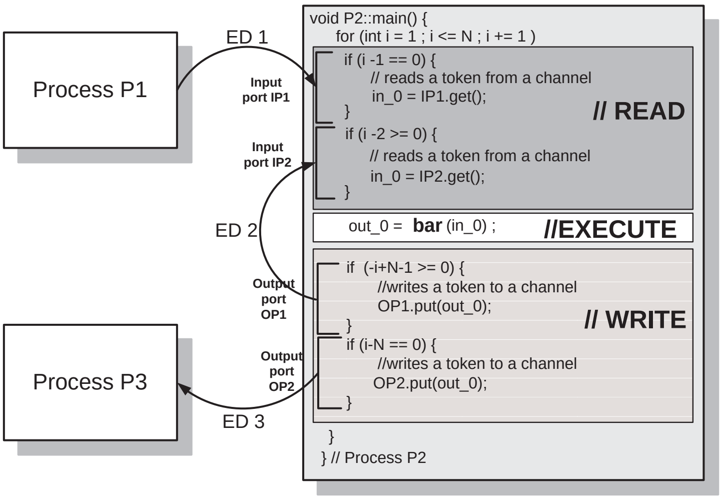


Figure 1.2: The CDFPN Network of the program listed in Listing 1.1

Consider the CDFPN shown in Figure 1.2. Like with most of the graphically programming environments, the nodes of the graph can be viewed as processes that run concurrently and exchange data over the arcs of the graph (communication channels). The given CDFPN consists of three processes, *P1*, *P2*, and *P3*, and three communication channels, *ED1*, *ED2*, and *ED3*. This graph is referred to as a network and it is the input-output parallel specification equivalent to the affine nested loop program listed in Listing 1.1. The network communication channels, *ED1*, *ED2*, and *ED3*, implements the global memory represented by vector *z(j)* in a distributed meaner. The network processes contain a subprogram that call a specific function of the affine nested loop program (i.e. *P1* calls `Init()` , *P2* calls `bar()` and *P3* calls `Sink()`). The network is constructed in such way that no control is shared between the processes. Hence, the functions from the affine nested loop program are also executed independently (i.e., no explicit synchronization signal is shared among them).

In a CDFPN, concurrent processes communicate only through one-way communication channels with unbounded capacity. The interface between a communication channel and a process is called a port. A port is either an *Input Port* or an *Output Port*. An Input Port is used for a read operation from a communication channel. An Output Port is used for a write operation to a communication channel. Each channel carries a sequence (a stream) that is made of atomic data objects called *tokens*. Each token is written (produced) exactly once, and read (consumed) possible more than once. Writes to the channels are nonblocking, but reads are blocking. This means that a process that attempts to read from an empty input channel stalls until the buffer has sufficient tokens to satisfy the read. Hence, when a process stalls the function will not evaluate. Each call of the function results in output tokens that are sent to the appropriate outgoing ports. The order in which a channel is read or written is given (i.e., schedule).

A process in the CDFPN model is a subprogram. This subprogram is a module wrapper



```
void P2::main() {
    for (int i = 1 ; i <= N ; i += 1 )
      if (i -1 == 0) {
          // reads a token from a channel
          in_0 = IP1.get();
      }
      if (i -2 >= 0) {
          // reads a token from a channel
          in_0 = IP2.get();
      }
      out_0 = bar (in_0) ;
      if  (-i+N-1 >= 0) {
          //writes a token to a channel
          OP1.put(out_0);
      }
      if (i-N == 0) {
          //writes a token to a channel
          OP2.put(out_0);
      }
   }
 } // Process P2
```

Figure 1.3: A CDFPN process unveiled

that isolates the computation (the affine nested loop function call) from the communication (the schedule according to which a channel is read or written). Such a subprogram is given in Figure 1.3 and shows the implementation of the process *P2*. The process has four ports: two input ports (i.e., *IP1* and *IP2*) and two output ports (i.e., *OP1* and *OP2*). We observe that the tokens written to *ED2* via *OP1* are read back by the same process via *IP2*. We refer to this kind of communication channel as a *self-loop*.

The schedule according to which a channel is read or written is given by the surrounding nested for-loop and if-statements. Each input port is guarded by if-statements that control the read from a communication channel. For example, when the for-loop iterator $i$ is equal to one, a token from the *IP1* input port is read. In all other cases a token from the *IP2* input port is read. The read token is placed in an internal temporary variable *in_0* that is used as an input argument for the function `bar`. The call of this function produces an output token stored in variable *out_0*. Next, this variable is written to one of the output ports. As in the case of the input ports, the output ports are also guarded by if statements. The if-statement gives the right order of writing tokens into network channels. For example, when the for-loop iterator $i$ is equal to $N$, then the Output Port *OP2* is active. In all other cases, the Output Port *OP1* is active.

Process *P2* reads either from channel *ED1* or from channel *ED2*, processes the token, and finally writes the processed token to either channel *ED2* or channel *ED3*, depending on loop iterator conditions. Hence, we can always distinguish three behavioral parts in any of our processes: the READ, the EXECUTE, and the WRITE parts. The part that reads tokens from communication channels via input ports is called the READ part, as shown in Figure 1.3. Operations on the read tokens take place in the EXECUTE part. In Figure 1.3, the EXECUTE part evaluates the `bar` function. The part that writes tokens to communication channels via output ports is called the WRITE part.

## 1.2 Problem Definition

Given an affine nested loop program and its input-output equivalent COMPAAN Data Flow Process Network specification, how can we implement that specification in an FPGA based multi-processor execution platform? In fact, this can be done in several ways. Current FPGA chips are powerful enough to allow heterogeneous architecture implementations. Thus, an architecture consisting of hardcore and/or softcore ISA components, configurable components, dedicated components, point-to-point, bus-based, or cross bar-based communication structures, and shared or distributed memory components, can be implemented in FPGA chips. This diversity of options has been investigated in [11]. However, the most efficient way to implement a CDFPN specification in an FPGA fabric is a dedicated one-to-one mapping of the former into the latter. Although this seems to be a straightforward approach, it is not so because the CDFPN processes are threads that read data, evaluate functions, and write data in a sequential order. When implemented as such, resource utilization and performance will not, and can not, be optimal.

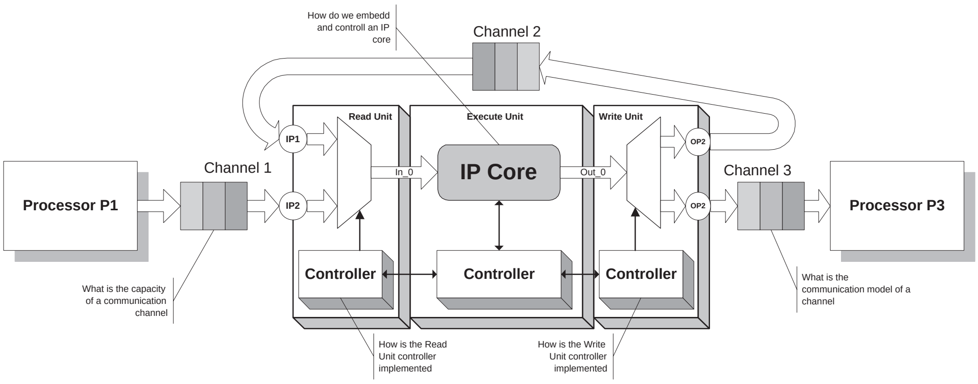

How do we embedd and controll an IP core
Channel 2
Read Unit
Execute Unit
Write Unit
IP1
IP2
In_0
IP Core
Out_0
OP2
OP2
Channel 1
Channel 3
Processor P1
Processor P3
Controller
Controller
Controller
What is the capacity of a communication channel
What is the communication model of a channel
How is the Read Unit controller implemented
How is the Write Unit controller implemented

Figure 1.4: Process network architecture example: The processor template

Considering the CDFPN example shown in Figure 1.2, then a possible implementation model and the issues to be addressed are depicted in Figure 1.4. In this model, a process is mapped to a processor. Each processor is decomposed into a *Read Unit*, an *Execution Unit*, and a *Write Unit* that operate in a pipelined fashion. The execute unit evaluates one or more functions that are enclosed in an intellectual property (IP) core. Function input arguments are delivered by the Read unit that selects the arguments from process input channels. Function results are delivered to the Write unit that distributes them across process output channels.

The main issue in mapping a CDFPN to an FPGA is *how to design an architecture such that it achieves the maximum data throughput for the given CDFPN*. Addressing this issue requires answering to the following questions:

- What is the *actual* capacity of a communication channel? The CDFPN MoC specifies that the intra processes communication is done over *unbounded* FIFOs, thus, we need to bound the channels capacities for an implementation. In [7], the communication channels are bounded to an over-dimensioned value. Hence, we have to determine the communication channel capacity so as to avoid memory spilling and network deadlocks.

- What are the actual communication primitives and protocols? Because the initial nested loop programs are streaming data based applications that enforce a throughput, the way in which the communication channel handles reading and writing operations should not obstruct the flow of data.
- Can the processor be always pipelined? If so, the CDFPN specification has to be transformed to exploit this option.
- How to embed an IP core in the processor template? The computation in a COMPAAN Process Network is not fully specified. Functions embedded in a process of a CDFPN are specified as mathematical functions $[out_0,\ out_1] = F(in_0,\ in_1,\ in_2)$, i.e., the implementation code is not included.
- How to implement the Read and Write units without hindering the performances of the IP core? The Read and Write units implement the blocking read and blocking write synchronization primitives. At each occurrence of a blocking read or write situation, the processors stall. Moreover, the Read and Write units have to determine the next read and write sequence at each clock cycle. Hence, the design of the Read and Write units is critical in obtaining an implementation that has a maximal data throughput.

## 1.3 Solution Approach

The problem of mapping a CDFPN to an FPGA is the subject of this thesis, resulting in a solution approach and an implementation called the LAURA tool. This solution is part of the COMPAAN/LAURA tool chain shown in Figure 1.5: the synthesis of applications specified as nested loop programs to an FPGA platform.

The mapping of a CDFPN in an FPGA consists of two parts. The first part is a *platform independent* step. In this step, a given CDFPN specification is converted into an *Abstract Architecture*. The Abstract Architecture is a set of hieratical interconnect modules representing an architecture. Each module isolates computation from communication. We call this platform independent step the *PN to Abstract Architecture* step, as no information of the actual targeted platform is taken into account. The second part is a *platform dependent* part that synthesizes an actual FPGA-based multi-processor execution organization from the Abstract Architecture. In this step, the Abstract Architecture is converted onto a *Network of Synthesizable Processors*. We call this the *Architecture Synthesis* step as platform specific information is taken in account. Also, we embed in this step platform specific IP cores that implement the Execute unit functionality of the synthesized CDFPN processes. The Network of Synthesizable Processors is specified using a hardware description language. In this thesis we limit ourselves to VHDL output.

The CDFPN specification, the Abstract Architecture, and the Network of Synthesizable Processors (the FPGA implementation) are topologically identical. However, their semantics are different. The semantics are related through a number of operations in the PN to Abstract Architecture and the Architecture Synthesis steps. The PN to Abstract Architecture step consists of the following operations:

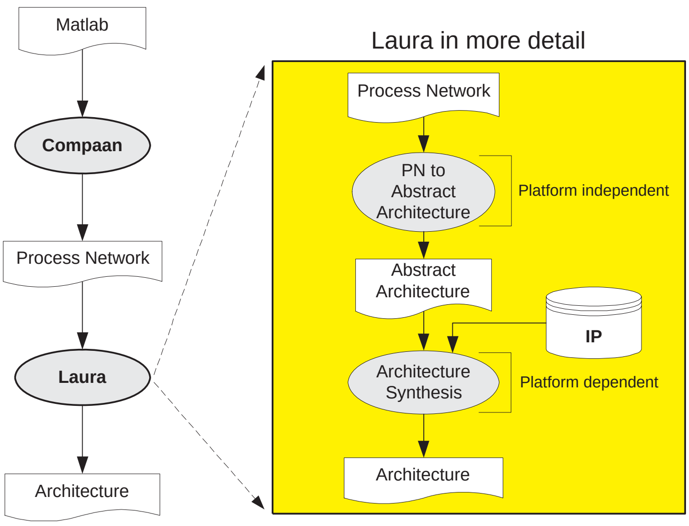


Figure 1.5: The LAURA flow in the COMPAAN/LAURA tool chain

- *Topological Mapping* converts the CDFPN to a network of virtual processors. The resulting network has the same topology as the CDFPN, as we employ one-to-one mapping in which each process becomes a *Virtual Processor* and each communication channel a *Dedicated Channel*;
- *Semantic Mapping* decomposes a PN process specification into the components of a Virtual Processor. These components are the *Read Unit*, the *Write Unit*, and the *Execute Unit*. The controller is distributed in the Read, Execute and Write units.

The Architecture Synthesis step consists of the following operations:

- *Control Synthesis* derives the control structure for the Read or Write units;
- *Communication Synthesis* determines the type and the capacity of a dedicated channel;
- *Expression Synthesis* translates a linear or pseudo linear expression to a form that is free of multiplication and integer division operations. The Expression Synthesis operation is used by the Control Synthesis operation;
- *IP Core Integration* embeds a functional IP core.

The Network of Synthesizable Processors is captured in the LAURA tool in terms of a set of both generic and platform specific VHDL templates. In this dissertation, we target specifically the VIRTEX II/Pro platform from Xilinx, as this platform was available to us. We use the Xilinx resources (e.g., embedded memories, serial interfaces) to implement the platform specific VHDL templates. When we embed an IP core in our network, a tailored processor is made to accommodate the IP core. The resulting processor inherits the execution model of the IP, the computational resources available, and the clock cycles needed for an execution.

## 1.4 Thesis Contribution

In this thesis, we present the LAURA approach that implements our methodology to map PNs generated by the COMPAAN compiler onto a reconfigurable platform such as an FPGA. The main contributions are:

- The development of an approach that allows mapping of a Process Network specification onto reconfigurable platforms in a systematic and automatic way;
- The introduction of the notions of Abstract Architecture, Virtual Processor and Dedicated Channel to capture and model the Process Network behavior on the FPGA;
- The development of a technique that improves the efficiency of the IP cores embedded in our architecture;
- The development of a technique that can estimate at *compile time* the type and capacity of the dedicated channels;
- The development of a number of techniques that allow us to map a CDFPN efficiently (in terms of speed and resources) onto an FPGA platform;
- The validation of the present approach with real-life industrial experiments;
- The prototyping of the present approach in software.

A number of experiments have been conducted for applications in the field of image processing and signal processing. The experiments show that we are able to fully automatically derive a FPGA implementation from a given sequential imperative application specification.

## 1.5 Related Work

Many researchers have addressed the problem of mapping sequential imperative programs to FPGA execution platforms [12–21]. In the literature, all contributions differ in the way programs and platforms are *constrained*. Mapping any sequential imperative program to an FPGA execution platform is almost equivalent to mapping such programs to homogeneous multi-processor architectures. Our approach is different as we generate FPGA implementations using a constrained Process Network MoC that uses the polyhedral model to derive the firing sequence of each process. In this section, we discuss some approaches that uses the Process Network MoC or the polyhedral model to generate hardware architectures. These approaches are discussed in two sections. The first section is dealing with the tools that generates an FPGA implementation using the polyhedral model. The second section deals with tools that use PN MoC to describe hardware architectures.

### 1.5.1 Hardware Architecture Implementations that uses the Polyhedral Model

The polyhedral model is used in numerous projects to synthesize architectures for multi-processor FPGA based architectures. The CLooGVHDL [22] project is one of them. Each

C statement processed by ClooG [23] is synthesized by the VHDL back-end using a sequential execution model. Thus, only the parallelism within one statement is exploited by the ClooGVHDL. In [24] Teich and Thile describe a systematic way to design a processor array. Although the processor arrays are a good solution to map applications that are data flow dominated, these processor arrays are not suited for more control dominated applications. Thus, the control dominated applications require a more complex global controller to be synthesized to. In the Paro compiler [25], the authors extend the work presented in [24] by a methodology which reduces the hardware cost of the global controller and memory address generators by avoiding costly multiplication and division operations.

In the Alpha environment [26], a program is described as a system of affine recurrence equations (SARE). Starting from such a specification, both the synthesis of regular architectures and the compilation to sequential or parallel machines are considered. The rationale behind writing programs in Alpha rather than in some imperative language is that a functional/mathematical specification matches the way people think of an algorithm and that all the parallelism in the algorithm is naturally preserved. AlpHard [27] is a subset of the Alpha language that enables the hardware generation of regular architectures, like systolic arrays.

Another example is the Atomium [28] project which consists of a set of tools that operate at the behavioral level of an application, expressed in C. The output is a transformed C description, functionally equivalent to the original program, but typically leading to strongly reduced execution times, memory size, and power consumption. Related to our work is the part of the Atomium dealing with memory issues when mapping applications onto platforms with distributed memory architectures. The Memory Architect is a component tool allowing the designer to explore the effects of timing constraints on the required memory architecture. This architecture translates the timing constraints into optimized memory architecture constraints: for a given set of timing constraints, it generates an optimized set of architectural constraints and a cost estimate for the resulting architecture.

The high-level synthesis methodology Phideo [29] starts with a specification in the single assignment form, and converts this description into an instance of a target architecture template. An important part of Phideo is the address generation method for memories that are introduced by the synthesis tool. The address generation in Phideo is a special case of the address generation present in the COMPAAN Data Flow Process Networks. This is due to a somewhat more restricted geometry of the iteration domain used in Phideo. The hardware designed by Phideo is synchronous and a schedule is derived. This represents an important constraint and therefore the class of applications Phideo accepts as input is restricted to single assignment perfectly nested loop programs.

The PICO project [30] at HP Labs (later on spun out to a startup called Synfora [31]) is an effort that aims to automate the mapping of applications onto platforms consisting of a VLIW processor and custom nonprogrammable accelerators (NPA) connected to a two-level cache subsystem connected to the system bus. Each accelerator is customized to execute a compute intensive loop nest that would otherwise have been executed on the VLIW. Different than in our network representation, an NPA is represented by a fixed size (non-parametric) array of processing units activated by a global schedule. The NPA is derived by the PICO-NPA compiler which accepts a perfect loop nest in C and produces, based on a template, a structural Verilog/VHDL that defines the NPA at the register transfer level together with the C code that repeatedly invokes the NPA hardware. This code is compiled onto the host processor along with the remainder of the application.

### 1.5.2 Hardware Architecture Implementations that uses the Process Network MoC

The usage of the Process Network MoC for FPGA implementation is not new. A special subset of the Process Network MoC called Communicating Sequential Processes (CSP) [32] has been used by a number of projects for modeling applications. The Stream-C project [33] lets the user specify coarse grain, process level parallelism. The compiler infers fine grain, loop level parallelism. Stream-C is also based on the CSP model and allows users to specify independent parallel processes and their mapping to a multiple FPGA platform. Another approach is to add constructs and annotations to a subset of a programming language to specify parallelism and event sensitivity. Examples of this approach include the Handle-C project at Oxford [34], where the compiler can produce hardware from an input description. Handle-C is based on Hoare's CSP model and it is a modified form of C, where the user can specify concurrent operations and bit-widths of data.

C-HEAP is a top-down design methodology presented in [35] using the Kahn Process Network (KPN) MoC. It generates instances of an architecture template containing dedicated hardware components, multiple software programmable processors (e.g., CPUs, DSPs), local cache memories, a global shared memory, and a communication network. Although the communication between various processors is made using KPN modeling, their hardware implementation is a bus oriented architecture. In this architecture, however, problems with the cache coherence are reported. In our approach we do not use global shared memory and thus memory contention is avoided.

In [36], the authors present a Kahn Process Network methodology based on the DISYDENT platform (DIgital SYstem Design ENvironmenT). The system is described by a set of communicating Kahn processes. These processes are C POSIX threads representing both software and hardware tasks. Each thread communicates with the others using channel-read/channel-write primitives. Systems realization consists of synthesizing hardware tasks to RTL-VHDL language. However, this methodology is suited for control dominated applications, requiring low level information to be given by the user.

The use of KPN as a model of computation is also reported in the COSY [37] and Prophid [38] tools. Prophid is a heterogeneous multi-processor architecture template. This template distinguishes between control-oriented tasks and high performance media processing tasks. A CPU connected to a central bus is used for control-oriented tasks and possibly low to medium performance signal-processing tasks. A number of application-specific processors implement high performance media processing tasks. These processors are connected to a reconfigurable high-throughput communication network to meet the high communication bandwidth requirements. Hardware FIFO buffers are placed between the communication network and the inputs and outputs of the application-specific processors to efficiently implement stream-based communication.

The COSY methodology [37] provides a gradual path for communication refinement in a top-down fashion for a given platform and a communication protocol. The main goal of the COSY methodology is to perform design space exploration of a system at a high abstraction level. In order to achieve this, the methodology provides a mechanism for modeling communication interfaces at a high level of abstraction (including the behavior of the selected protocol) with various parameters, e.g. delay of execution of the protocol itself. The COSY methodology uses a message passing communication protocol with read and write primitives.

The architecture generated by us has close ties with the Globally Asynchronous Locally Synchronous systems [39]. GALS systems contain several independent synchronous blocks which operate with their own local clocks and communicate asynchronously with each other. The main feature of these systems is the absence of a global timing reference and the use of several distinct local clocks (or clock domains), possibly running at different frequencies. However, we know of no tools that generate GALS systems out of a imperative program like our proposed COMPAAN/LAURA approach does.

## 1.6 Thesis Outline

The general outline for the LAURA approach is shown in Figure 1.6.

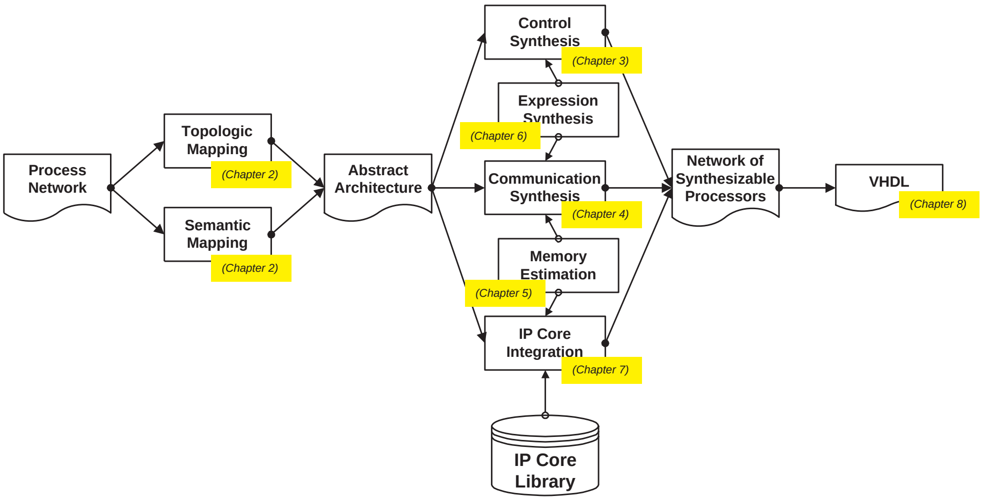


Figure 1.6: The Laura Flow

In Chapter 2, we present the step of mapping a process network onto an Abstract Architecture. In Chapter 3, we present our methodology to map the associated control for each Read and Write unit of the Virtual Processor. We pay attention to constraints such as clock speed and area in the mapping of those units. In Chapter 4, we analyze the communication behavior between processors and propose four channel realizations. In Chapter 5, we present a methodology to determine at compile time the memory requirements for a particular communication channel and thus, for the entire network. In Chapter 6, we present our methodology to synthesize complex expressions which are found in the control units. The topic of IP core integration is discussed in Chapter 7. Here we present how an IP core is embedded into our Virtual Processor and how we can determine its utilization in the case of pipelined IP cores in the presence of self-loops. In the case of a low utilization of the IP pipeline, we propose a number of transformations that may increase this utilization. In Chapter 8 we present three software kernels which are used in smart antenna applications. We conclude the thesis in Chapter 9.

Chapter 2

# From COMPAAN Data Flow Process Network to Abstract Architecture

In this chapter, we deal with the first two operations that are present in the first step of the LAURA approach, that is called *PN to Abstract Architecture*, see Figure 2.1. These two operations are the *Topological Mapping* and the *Semantic Mapping*. The Topological Mapping translates a CDFPN network topology to an architectural communication network. The Semantic Mapping structures each process of the network to a form that is suitable for synthesis. The core of this structure is the Read-Execute-Write(REW) synthesis template. The result specifies the CDFPN model in architectural terms as an *Abstract Architecture*. Before we further explain the PN to Abstract Architecture step, we first provide some background information in Section 2.1 concerning the model of computation used. In Section 2.2, the Topological Mapping relates the topology of the COMPAAN Data Flow Process Network (MoC) with the topology of the Abstract Architecture. In Section 2.3, we discuss about the *Semantic Mapping* which translates the behavior of the processors in the CDFPN MoC to the behavior of the processors in the Abstract Architecture. The chapter is concluded in Section 2.4.

## 2.1 Background

The model of computation (MoC) that we consider in this thesis is a process network model which we call it COMPAAN Data Flow Process Network (CDFPN). An CDFPN is a special case of the Kahn Process Network [1, 2] MoC, sharing many of the characteristics of KPN MoC (e.g., the communication semantics of CDFPN is the same as for Kahn networks). A Kahn Process Network (KPN) consists of a set of processes that communicate point-to-point over unbounded FIFO channels. A process that wants to read from a channel will block when that channel is empty waiting for data to arrive. Compared to Kahn networks, a COMPAAN network does not allow a process or a network to be *nondeterministic*, and the network is always static (i.e., no additional nodes or arches can be added at run-time). Also, the hierarchical characteristic of a generic KPN is not kept by CDFPN MoC. However, KPN

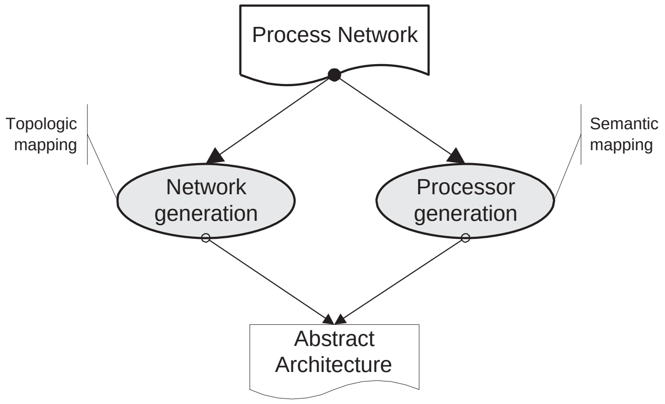


Figure 2.1: PN to Abstract Architecture step in more detail

has the following favorable characteristics shared also by CDFPN networks:

- The KPN model is deterministic, which means that irrespective of the schedule chosen to evaluate the network, always the same input/output relation exists;
- The inter-process synchronization is done by a blocking read. This is a very simple synchronization protocol that can be realized easily and efficiently in FPGAs;
- Processes run autonomously and synchronize via the blocking read. When mapping processes to an FPGA, you get autonomous islands on the FPGA that are only synchronized via blocking reads;
- As control is completely distributed to the individual processes, there is no global scheduler present. As a consequence, partitioning a KPN over a number of reconfigurable components or microprocessors is a simple task;
- As the exchange of data has been distributed over the FIFOs, there is no notion of a global memory that has to be accessed by multiple processes. Therefore, no resource contention occurs.

Listing 2.1: Static Affine Nested Loop Program example

```
for i = 1 : 1 : M,
   for j = 1 : 1 : N,
     if i+j <= T,
       [r(i,j)] = F(...);
     end
   end
end
```

Our model is special in that the processes are static affine nested loop programs (SANLP). In an SANLP, the loop bounds, condition statements, and variable indexing functions are all affine or pseudo affine functions of loop iterations and static parameters. A static parameter is a parameter that is constant during a SANLP execution. An example is shown in Listing 2.1 in which $M$, $N$, and $T$ are static parameters, $i$ and $j$ are the index variables of the surrounding loop. Hence, $Z(i, j)$ and $Z(i, i + j)$ are affine accesses. A function of one or more variables, $i_1, i_2, \ldots, i_n$ is *affine* if it can be expressed as a sum of a constant, plus constant multiples of the variables, i.e., $c_0 + c_1x_1 + c_2x_2 + \ldots + c_nx_n$, where are $c_0, c_1, \ldots, c_n$ constants.

In CDFPN networks, the firing rules are derived in a particular manner that respects the SANLP nature of the COMPAAN sequential imperative input programs:

- A firing rule dictates how the tokens are consumed in one process fire.
- A process interacts with the rest of the network *only* through their FIFO links. A CDFPN is input-output equivalent to a SANLP, and can be automatically derived from that equivalent program by the COMPAAN compiler. Internally, the COMPAAN compiler uses a representation of loops so-called the polyhedral program model [7, 9, 10].

It is important to understand what COMPAAN compiler generates a CDFPN from a SANLP input program. Consider the program shown in Listing 2.2. This program uses the Matlab programming language as this is the input programming language accepted by the COMPAAN compiler.

Listing 2.2: SANLP compilation example

```
for j = 1 : 1 : N,
   for i = 1 : 1 : N,
     [x(j, i)] = $F_1$(...);
     if i+j <= N,
         [] = $F_2$(x(j, i));
     end
   end
end
```

The above program can be represented using the polyhedral program model, where each statement is guarded by a set of linear equations. A *statement* is a line of a program without control (e.g., [] = $F_2$(x(j, i)) from Listing 2.2). A statement is executed for a set of values of the *iteration* vector, the vector containing the iterators of surrounding loops (i.e., for statement containing the function $F_2()$, the iteration vector is $(i, j)$). The *iteration domain* is the set of values of the iteration vector for which the statement is executed. The iteration domain of the statement containing the function $F_2()$ is shown in Figure 2.2.

The program listed in 2.2 is equivalent to the process network presented in Figure 2.3. The network is made out of two processes called *producer* and *consumer*. The producer process wraps the function call of $F_1()$ of given SANLP example and the consumer process wraps the function call of $F_2()$ of given SANLP example. The producer process is characterized by the C program:

```
for ( int $j_1 = 1$ ; $j_1 <= N$ ; $j_1 += 1$) {
  for ( int $i_1 = 1$ ; $i_1 <= N$ ; $i_1 += 1$) {
    $out_0 = F_1()$;
    if ($-j_1 - i_1 + N >= 0$) {
      $OP_1$.put( $out_0$ );
    }
  }
}
```

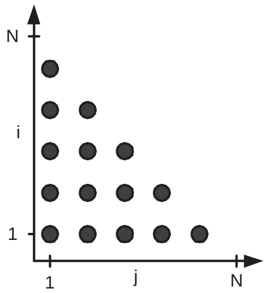


Figure 2.2: The graphical representation of the iteration domain of statement containing the function $F_2()$ from Listing 2.2

And the consumer process is characterized by the C program:

```
for ( int $j_2 = 1$ ; $j_2 <= N - 1$ ; $j_2+ = 1$) {
  for ( int $i_2 = 1$ ; $i_2 <= -j_2 + N$ ; $i_2+ = 1$) {
    $in_0 = IP_1$.get();
    $F2(in_0)$;
  }
}
```

The communication between these two processes is done using an unbounded FIFO buffer (i.e., $ED_1$). The interface between the FIFO buffer and the producer process is denoted by a black point (i.e., $OP_1$). This interface is called the *Output Port*. An Output Port is used for a write operation to a FIFO. The interface between the FIFO buffer and the consumer process is denoted also by a black point (i.e., $IP_1$). This interface is called the *Input Port*. An input port is used for a read operation from a FIFO.

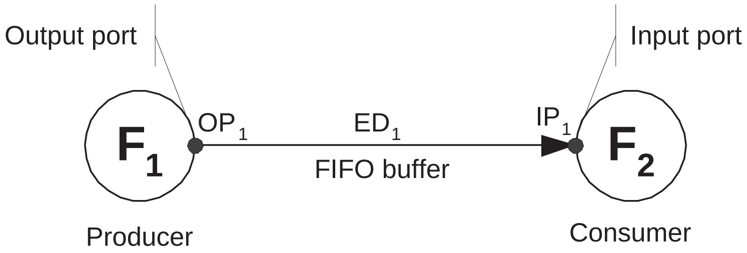


Figure 2.3: CDFPN network of the SANLP presented in Listing 2.2

The two processes from Figure 2.3 and their relation can be represented by a *producer-consumer pair* P/C pair) [40] and an affine mapping.

**Definition 2.1** A P/C pair is a tuple $< \mathcal{C}(p), f, P(p), \prec >$, where $\mathcal{C}(p) \subset \mathbb{Q}^n$ is a parameterized polytope, $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$ is an affine function, $\mathcal{P}(p) = f(\mathcal{C}(p) \cap \mathbb{Z}^n)$, and $\prec$ is the lexicographical order.

Thus,

$$\mathcal{C}(p) = \{x \in \mathbb{Q}^n \mid Ax + Bp \geq C\}, \tag{2.1}$$

with $A$, $B$, and $C$ integral matrices of appropriate dimension, and $p$ static integral parametric vector. $\mathcal{C}(p) \cap \mathbb{Z}^n$ is called the *consumer domain*. A consumer domain is an iteration domain.

$$\mathcal{P}(p) = f(\mathcal{C}(p) \cap \mathbb{Z}^n) = \{i \in \mathbb{Z}^m \mid i = Mk + O \wedge \ k \in (\mathcal{C}(p) \cap \mathbb{Z}^n)\}. \tag{2.2}$$

with $M$ and $O$ integral matrices of appropriate dimension. Usually matrix $O$ is zero and matrix $M$ is called mapping matrix. $\mathcal{P}(q)$ is called the *producer domain*. A producer domain is an iteration domain.

In our example, the mapping function $f$:

$$f\begin{pmatrix} j_1 \\ i_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j_2 \\ i_2 \end{pmatrix}$$

maps the points $(j_2, i_2)$ from the consumer domain

$$\mathcal{C} \cap \mathbb{Z}^2 = \{ \begin{pmatrix} j_2 \\ i_2 \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} j_2 \\ i_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} [N] \geq \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \}$$

to the points $(j_1, i_1)$ from the producer domain

$$\mathcal{P} = \{ \begin{pmatrix} j_1 \\ i_1 \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} j_1 \\ i_1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} [N] \geq \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \}$$

The lexicographic order ”$\prec$” is the order as defined through the loop nests. The lexicographic order is also referred as the *local schedule*. Hence, the local schedule of the producer process is given by the following nested for-loops:

```
for ( int j1 = 1 ; j1 <= N ; j1+ = 1)
  for ( int i1 = 1 ; i1 <= N ; i1+ = 1)
```

and the local schedule of the consumer process is given by the following nested for-loops:

```
for ( int j2 = 1 ; j2 <= N − 1 ; j2+ = 1)
  for ( int i2 = 1 ; i2 <= −j2 + N ; i2+ = 1)
```

The understanding of P/C pair is essential as a CDFPN is a collection of these P/C pairs arranged in a network [7]. In [41], the authors present four types of communication that can exist in a P/C pair. The four types differ in order and multiplicity. *Multiplicity* means that a token that is sent by the producer is read more than once at the consumer side. *Order* means that a token sent by the producer is not read by the consumer in the same order as it was produced. Depending on the order and presence of multiplicity, an arbitrary communication channel belongs to one of the following four classes:

**In-Order without multiplicity (IOM-)** a producer writes data in the channel in the same order and quantity as the consumer reads from the channel;

**In-Order with multiplicity (IOM+)** the order in which data is produced is the same as the order in which data is consumed. However, some data is consumed more than once, breaking the communication model of a FIFO, where a get operation is destructive. In this model, the life-time of a token needs to be taken into account;

**Out-of-Order without multiplicity(OOM-)** a consumer reads data in a different order than it has been written by the producer;

**Out-of-Order with multiplicity(IOM+)** a channel has the same characteristics as in the Out-of-Order case. Additional release logic is added at the consumer side to keep track of the life-time of tokens, to determine the release moment.

To synthesize a CDFPN to an FPGA multi-processor execution platform, we need a model that facilitates this synthesis. This model is the *Abstract Architecture*. It is defined in terms of concurrent autonomous *Virtual Processors* (VP) that communicate in a point to point fashion over *bounded* channels. In the next sections we show how a CDFPN is mapped into an Abstract Architecture. The mapping to an Abstract Architecture is made using two operations. The first operation, generates an interconnection network between a number of processors. The processors are generated by the second operation. In this second operation, we make use of a predefined processor architecture template. Due to the fact that the first operation deals with topologies of an architecture, this first operation is called *Topological Mapping*. The second operation is called *Semantic Mapping*. The semantic model of a processor is defined in this second operation, i.e., how the various components interact with each other; autonomous processors that communicate over channels using blocking read and blocking write semantics.

## 2.2 Topological Mapping

The topological mapping operation creates an architectural interconnection network that has the same topology as the topology of the given CDFPN MoC. This is due to one-to-one topological mapping used by our methodology. The architectural interconnection network is captured by our *Abstract Architecture* model. The one-to-one mapping ensures that the task level parallelism (TLP) captured by the CDFPN MoC is propagated to the FPGA level. The Abstract Architecture *naturally fits* the CDFPN MoC when:

- The Abstract Architecture communication and synchronization primitives match the CDFPN MoC communication and synchronization primitives;
- The operational semantics of the Abstract Architecture match the operational semantics of the CDFPN MoC;
- The data types used in the CDFPN MoC match the data types used in the architecture.

Hence, the Abstract Architecture interconnection network has *at least* the following implementation characteristics:

- Communication and Synchronization:
    - FIFO buffer used to realize the interprocess communication;
    - Blocking read synchronization primitive is part of a FIFO behavior. Any virtual processor implements the read token from FIFO function as an execution blocking function.
- Operational semantics. An FPGA multi-processor execution platform provides a large number of opportunities to create various number of complex operations. These operation can be realized using the embed CPUs or DSP blocs, distributed block RAMs, input/output blocs and Configurable Logic Blocs (CLBs).
- An FPGA multi-processor execution platform is optimized to work with scalar data types rather with packages of data. Hence, in this dissertation, we consider that a CDFPN token (or datum) is always a scalar.

Additionally, due to the fact that a FIFO buffer is bounded in real life, a new synchronization primitive is introduced. This is the *blocking write* synchronization primitive. This protocol states that a virtual processor halts when it attempts to write to a channel that is full. Thus, a virtual processor can continue its execution whenever there are neither blocking read nor blocking write situations.

Not all the communication channels of the Abstract Architecture are FIFOs as in the case of the CDFPN. This is due to a different approach in handling the four types of communication that exists in CDFPN. These communication types has been introduced in [41] and enumerated in Section 2.1. This derogation does not change the topology of the Abstract Architecture in respect to the topology of the COMPAAN generated PN. However, the communication channels may have differen execution behavior than a FIFO (First In First Out) (e.g., LIFO - Last In First Out - execution behavior, circular buffers).

Our philosophy is to keep a virtual processor of an Abstract Architecture memoryless, and all the communication related memory is handled by the Abstract Architecture interconnection network. Thus, we clearly separate the *computation* from *communication* as a virtual processor only deals with the computational part of the CDFPN MoC and the Abstract Architecture interconnection network with the communication. The different communication primitives that the Abstract Architecture interconnection network employs are discussed in Chapter 4. The output ports and the input ports of a CDFPN are instantiated accordingly to the type of channel used. An example of an abstract interconnection network is the CDFPN given in Figure 2.3. However, the FIFO buffer is bounded in Abstract Architecture interconnection network. We discuss in Chapter 5 how to determine the upper limit of all the channels types used in an implementation. More complex network examples are given in the next sections and chapters.

## 2.3 Semantic Mapping

We showed before how the topology of the Abstract Architecture is generated using the topological mapping operation. This operation defines the communication channels in terms of types, bounds and synchronization primitives. Each of these communication channels is connected to a processor. This processor is called *Virtual Processor* (VP). In this section, we show what is the synthesis template of a VP, and how to properly synthesize a VP into an Abstract Architecture.

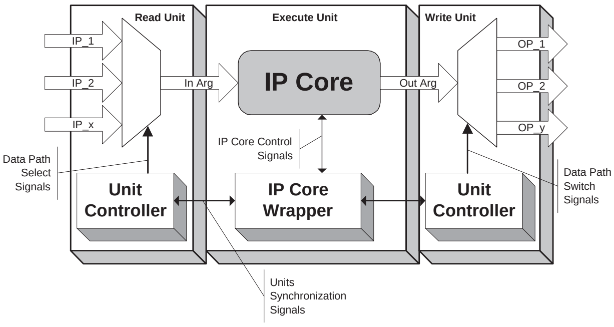


Figure 2.4: The Virtual Processor synthesis template

We start presenting the synthesis template of a VP in Figure 2.4. A virtual processor is composed of three units: a *Read Unit*, a *Write Unit*, and an *Execute Unit*. The Execute unit is the computational part of a virtual processor. It has a number of *input arguments* that provide to the unit the necessary data for execution and a number of *output arguments* that are the result of the computation process. In our model of implementation, the Execute unit fires when all the input arguments have data and always produces data to all the output arguments at once. The functionality of the Execute unit is realized using an IP core. The IP core is taken from an IP library. The control of the execution unit firing and of the embedded IP core is done by IP core wrapper. The IP core wrapper is discussed in Chapter 7. The Read unit is responsible for assigning all the input arguments of the Execute unit with valid data. Since there are more input ports than arguments, the Read unit has to select from which port to read data. This selection is realized by a *selector*. The selector is controlled by the local unit controller. An *Input Port* is the input interface that connects the virtual processor with a communication channel. An Input Port is characterized by an iteration domain called *Input Port Domain* (IPD). The *Output Port* is the output interface that connects the virtual processor with a communication channel. An Output Port is characterized by an iteration domain called *Output Port Domain* (OPD). The Write unit is responsible for distributing the results of the Execute unit to the relevant processors in the network. A write operation can be executed only when all the output arguments of the execute unit are available for the write unit. Since there may be more output ports than output arguments, a *switch* is used for the proper selection of the Output Port. The switch is controlled by a local unit controller. The local unit controller selects the proper Output Port accordingly to the current state (i.e., iteration) of the virtual processor.

A key characteristic of the VP is that it models an overlay of these three units realized using inter-units synchronization signals. The overlay execution model of a VP is shown in Figure 2.5, where **R** is the execution of the Read unit, **E** is the execution of the Execute unit, and **W** is the execution of the Write unit.

The processes in a CDFPN are static affine nested loop programs. These programs are mapped by semantic mapping operation to a VP. Thus, the Von-Neumann model of execution

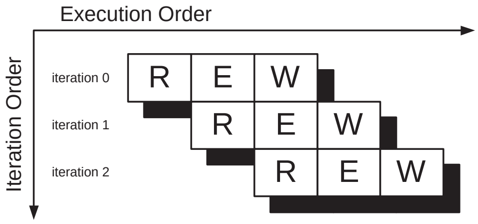


Figure 2.5: The pipelined model of execution of a virtual processor

of each process is optimized to fit the overlay operation mode of a VP synthesis template. Moreover, the operations made in Read unit and Write unit are parallelized to exploit the parallelism of the underlaying FPGA architecture as discussed in Chapter 3. The semantic mapping operation just fills the data structure of each VP unit with the required information. The synthesis of each unit is handled latter on in our methodology.

Consider the simple example show in Chapter 1, and reproduced in Listing 2.3. The CDFPN of this example is shown in Figure 2.6. As shown in the pervious section the example topology of the Abstract Architecture is the same as the topology of the CDFPN shown in Figure 2.6.

Listing 2.3: A simple Matlab program

```
for i = 1 : 1 : 1,
  [ z(i) ] = Init();
end

for i = 1 : 1: N,
  [z(i+1)] = bar(z(i));
end

for i =N+1 : 1 : N+1,
  [] = Sink(z(i));
end
```

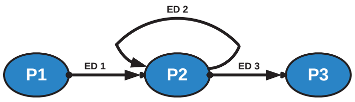
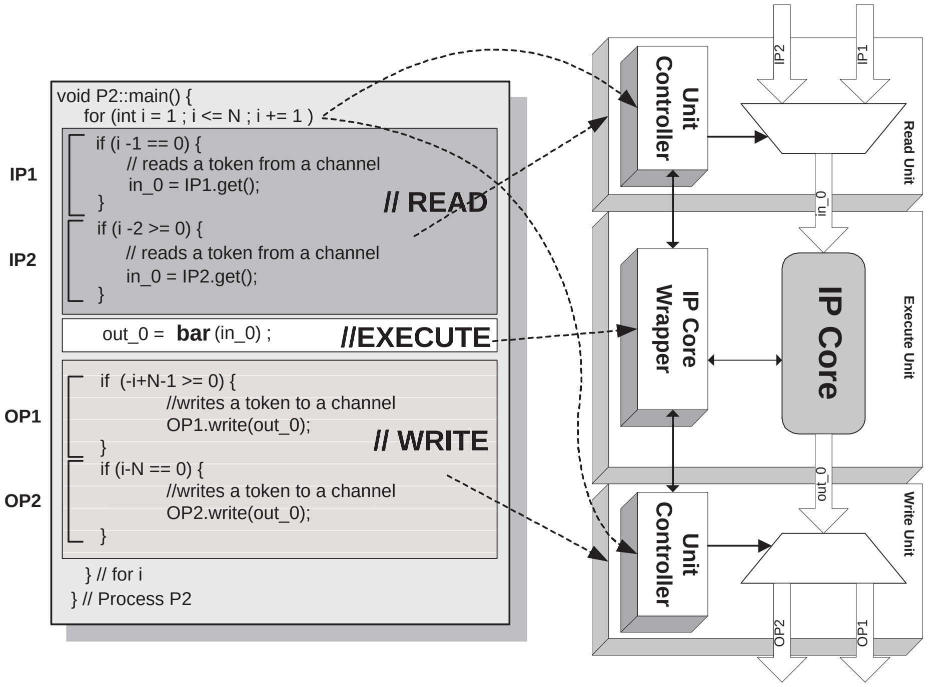


Figure 2.6: The CDFPN Network of the simple Matlab Program

Let us see how the semantic mapping fills up the VP synthesis template structure for the virtual processor that corresponds to the $bar()$ function. We observe that the function $bar()$ is wrapped by the COMPAAN compiler into an equivalent process able to run the autonomous static affine nested loop program shown at the left of Figure 2.7. This program reveals three phases: a read phase, an execute phase, and a write phase. The for-loop represents the local schedule of the process. In the read phase, the argument $in\_0$ of the function $bar()$ is read from either the Input Port $IP1$ or the Input Port $IP2$, depending whether $i = 1$ or $i \geq 2$, respectively. In the execution phase, the function $out\_0 = bar(in\_0)$ is evaluated. Finally, in the write phase, the result $out\_0$ is written to either the Output Port $OP1$ or the Output Port $OP2$, depending on whether $(i \leq N - 1)$ or $(i = N)$, respectively.

Clearly, the evaluation of the conditions that distinguish between input ports $IP1$ and $IP2$, and output ports $OP1$ and $OP2$ can be performed concurrently. In addition to that,



```
void P2::main() {
  for (int i = 1 ; i <= N ; i += 1 )
    if (i -1 == 0) {
         // reads a token from a channel
         in_0 = IP1.get();
    }
    if (i -2 >= 0) {
         // reads a token from a channel
         in_0 = IP2.get();
    }
      out_0 = bar (in_0) ;
    if  (-i+N-1 >= 0) {
              //writes a token to a channel
              OP1.write(out_0);
    }
    if (i-N == 0) {
              //writes a token to a channel
              OP2.write(out_0);
    }
  } // for i
} // Process P2
```

Figure 2.7: Semantic Mapping: The creation of a LAURA Processor

reading $in\_0$ from one of the two input ports, evaluation the function $out\_0 = bar(in\_0)$, and writing $out\_0$ to one of the two output ports can be overlapped in time. Hence, the semantic mapping operation fills up the VP structure as follows:

**Read Unit** The Read unit structure is filled up with the nested for-loop program that corresponds to the read phase of the mapped process. This program is evaluated at run-time by the Read unit controller at each firing of the embedded function. The selector of the Read unit is realized as a function of the number of input ports of the processor, and the number of input arguments of the embedded function. In our example, the selector is synthesized as 2-to-1 selector and the Read unit structure is filled with the following nested for-loop program:

```
for (int i = 1; i <= N; i++)
{
  if (i-1 == 0) {
    in_0 = IP1.get();
  }
  if (i-2 >= 0) {
    in_0 = IP2.get();
  }
}
```

**Write Unit** The Write unit structure is filled up with the nested for-loop program that corresponds to the write phase of the mapped process. This program is evaluated at run-time by the Read unit controller at each firing of the embedded function as in the case of the Read Unit. The switch of the Write unit is realized as a function of the number of output ports of the processor, and the number of output arguments of the embedded function. In our example, the switch is synthesized as 1-to-2 distributor and the Write unit structure is filled with the following nested for-loop program:

```
for (int i = 1; i <= N; i++)
{
  if (-i+N-1 == 0) {
    OP1.write(out_0);
  }
  if (i-N == 0) {
    OP2.write(out_0);
  }
}
```

**Execute Unit** The Execute unit embeds an IP core that is extracted from a given IP library. The Execute unit structure is filled with the required information to identify and embed an IP core into a VP processor. Such information is the function name (i.e., $bar()$) and the number of input and output arguments of this function. The number of input arguments and output arguments of the Execute unit is an exact match of the process function input and output arguments.

The execution of Read, Execute and Write units is synchronized using the units synchronization signals. The semantic mapping operation is represented graphically in Figure 2.7.

## 2.4 Conclusions

In this chapter we presented the COMPAAN Data Flow Process Network MoC as it is provided by the COMPAAN compiler. We observed that CDFPNs are derived from SANLP programs that can be analyzed using the polyhedral model. We showed how we map a CDFPN into a synthesis ready structure that we call Abstract Architecture. As a consequence, the polyhedral model characteristics are propagated downwards to the Abstract Architecture at the FPGA level. Central to this architecture is the Virtual Processor model. This model is tuned to take advantage of the inherited polyhedral model of CDFPN. In Chapter 3, we show how we take advantage of the polyhedral program model to generate local controllers for the Read and Write units. Optimizations of these local controllers are investigated more in detail in Chapter 6. The issues of creating a suitable communication structure between the various VPs is discussed in Chapter 4 and Chapter 5. Adding the computation part to our network is done via embedding IP cores. The issue of embedding IP cores and how they are interfaced with the Read unit and Write unit controllers is handled in Chapter 7.

# Chapter 3

# Control Synthesis

In the pervious chapter, we considered the conversion of a CDFPN to an Abstract Architecture (AA). Within the Abstract Architecture, we defined the virtual process (VP) that consist of Read, Write, and Execute units as shown in Figure 3.1. Read and Write units have an *unit controller* that selects at a specific iteration the FIFOs to read from and to write to, respectively. In this chapter, we specify these control units as a result of *Control Synthesis* operation. We first explain in Section 3.1 what control synthesis is. The control units of the Read or Write units can be implemented in three different ways, each having its own limitations and rewards. In Section 3.2 we consider the first strategy which is based on the usage of look-up tables. In Section 3.3, we present the parameterized predicate controller approach. In Section 3.4, the partitioned controller is discussed.We conclude this chapter in Section 3.5.

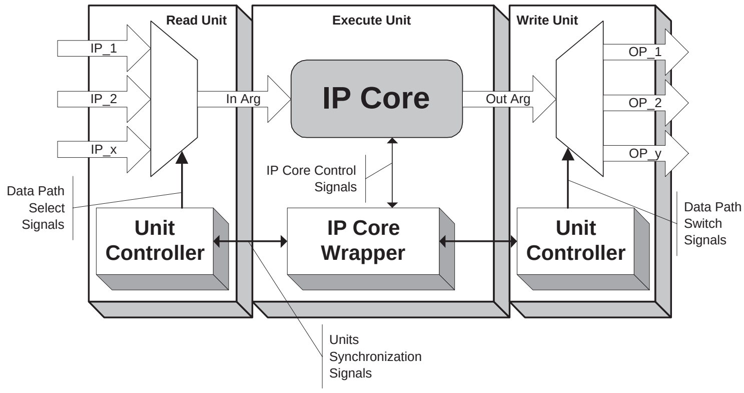


Figure 3.1: The Virtual Processor template

## 3.1 Control Synthesis in Read/Write Control Units

The Abstract Architecture's Virtual Processor template is shown in Figure 3.1. It consists of a Read unit, an Execute unit, and a Write unit. The Execute unit consists of a function $[\{out\_args\}] = F(\{in\_args\})$ that is implemented in an IP core. The IP core is controlled by an IP core wrapper. The input arguments $in\_arg$ for the IP core are delivered by Read unit that is essentially a select unit. This select unit selects one or more input channels from a set of input channels. An input channel is only selected if its FIFO contains an input argument for the function in the Execute unit. Input channel selector is under the control of the Read unit controller.

The output arguments $out\_args$ of the IP core are sent to the Write unit that is essentially a switch unit. It selects one or more output channels from a set of output channels. An output channel is only selected if an output argument of function in the Execute unit is to be written to the channels's FIFO. Output channel selection is under the control of the Write unit controller. The three units Read unit, Execute unit, and Write unit are synchronized for correct behavior of the Virtual Processor.

Recall that a port of the Virtual Processor's Read unit or Write unit is characterized by a port domain and a lexical ordering. These domains are sub-domains of the Execute unit's function domain. The Read unit and Write unit control units take care of the selection of the appropriate port domains and the order of reading arguments from and writing arguments to the corresponding channels.

Let us exemplify how an unit controller works by considering the example given in Listing 3.1.

Listing 3.1: Nested loop program

```
for i = 1 : 1 : 5,
   for j = i : 1 : 5,
      a(i+j) = F1(a(i+j));
   end
end
```

The function $F1$ has one input argument and one output argument. A data for the input argument of the function $F$ is read from one of Virtual Processor's input ports. The Virtual Processor corresponding to the $F1$ statement has three input ports (i.e., $IP_1$, $IP_2$, and $IP_3$). The iteration domains for each Input Port is show as follows:

$$IPD_1 = \{(i,j) \in \mathbb{Z}^2 \mid i = 1,\ 1 \leq j \leq 5\} \tag{3.1}$$

$$IPD_2 = \{(i,j) \in \mathbb{Z}^2 \mid 2 \leq i \leq 5,\ j = 5\} \tag{3.2}$$

$$IPD_3 = \{(i,j) \in \mathbb{Z}^2 \mid 2 \leq i \leq 4,\ i \leq j \leq 4\} \tag{3.3}$$

The Execution unit's function iteration domain is:

$$\mathcal{D} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 5, i \leq j \leq 5\}$$

The lexical order in $\mathcal{D}$ is the Ehrhart polynomial [42] rank:

$$k = -\frac{1}{2} * i^2 + (N + \frac{1}{2}) * i + j - N, N = 5$$

The graphical polyhedral representation of these iteration domains is show in Figure 3.2.

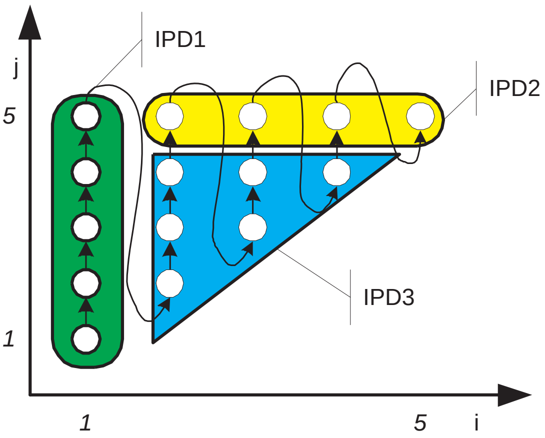


Figure 3.2: Polyhedral representation of the Input Port Domains of function $F1$. The arrows indicate the execution order

The Read unit port selection mechanism can be represented as follows:

```
for (int i = 1 ; $i \le 5$; i++){
  for (int j = 1 ; $i \le 5$ ; j++) {
     if ($i == 1$)
     {
       $in_0 = IP_1$.get(); // Read from Channel 1
     }
     if ($2 \le i \le 5 \land j == 5$)
     {
       $in_0 = IP_2$.get(); // Read from Channel 2
     }
     if ($2 \le i \le 4 \land i \le j \le 4$)
     {
       $in_0 = IP_2$.get(); // Read from Channel 3
     }
   }
}
```

Table 3.1 shows an ON/OFF relation between the port domains for all pairs $(i, j) = 1 \le i \le 5, i \le j \le 5$.

The unit controller of Read or Write units is the most critical component in a Virtual Processor. Its implementation must be fast enough to not obstruct the flow of data. Although the Read unit and Write unit controllers can be implemented as basic FPGA multiplexors and de-multiplexors, their operational behavior depends on the way how the Read or Write unit local controller is implemented. There are three approaches to synthesize a unit controller. They are:

- Look-up table controller
  In this approach the control sequence of an unit controller as depicted in Table 3.1 is realized using a look-up table. The entry of the look-up table is a current execution

| i | j | $IP_1$ | $IP_2$ | $IP_3$ |
|---|---|---|---|---|
| 1 | 1 | on | off | off |
| 1 | 2 | on | off | off |
| 1 | 3 | on | off | off |
| 1 | 4 | on | off | off |
| 1 | 5 | on | off | off |
| 2 | 2 | off | off | on |
| 2 | 3 | off | off | on |
| 2 | 4 | off | off | on |
| 2 | 5 | off | on | off |
| 3 | 3 | off | off | on |
| 3 | 4 | off | off | on |
| 3 | 5 | off | on | off |
| 4 | 4 | off | off | on |
| 4 | 5 | off | on | off |
| 5 | 5 | off | on | off |

Table 3.1: The activation table for each $IPD$ in relation with the $F1$ schedule.

iteration of $F1$ (e.g., $< i, j >$). The result of the look-up table is the proper selection of the corresponding $IPDs$ or $OPDs$.

- Parameterized predicate controller
  In this approach the control sequence of an unit controller is evaluated at run-time using the IPD/OPD mathematical descriptions (e.g.,$IPD_1 = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq j \leq 5,\ 1 \leq i \leq 1\}$).
- Partitioned parameterized predicate controller
  In this approach the control sequence of an unit controller is evaluated in a similar manner as in the case of the Parameterized predicate controller. However, we optimize the synthesis of an unit controller for a fast execution of parts of the control sequences that depend on the most inner loop index (e.g, for $IPD_1$ we have $1 \leq j$ and $j \leq 5$). The other parts are evaluated using a sequencer.

In the next sections we detail these approaches. We focus on the realization of Input Port control. The Output Port control is mutatis mutandis the same.

## 3.2 The Look-up Table Controller

The accepted input of the COMPAAN tool is statical parameterized nested for loop algorithms. The COMPAAN tool fully automates the transformation of the input Matlab code into CDFPNs, transferring the static characteristic to the network control. Because of this static characteristic, the sequence of reading/writing data from/to the FIFO channels is data-independent. The processes in a CDFPN are (possibly endlessly repeating) static sequences of Read-Execute-Write operations. Hence, a simple approach to implement the local schedule of a Virtual Processor is to use read only memory (ROM) tables to store the IPD/OPD activation sequences for a process domain iteration. This is equivalent to tracing the execution of a VP and to store the IPD/OPD sequences in ROM (see Table 3.1).

A trace for an IPD or an OPD is a list that contains the activation sequence of the port. The length of the list is equal to the number of integral points contained in the iteration port domain. Domains IPDs and OPDs are subsets of the process iteration domain that are scanned in lexical order. Larger domains generates larger traces. To overcome this problem, it is possible to apply compression algorithms and perform on-the fly decompression. In the COMPAAN/LAURA context, we must limit ourselves to very simple compression techniques, such as Run Length Encoding (RLE), since we want the decompression engine area overhead to be very limited and not to cause additional delays.

The architecture constrains require a new activation value to be ready within one cycle, which can be easily achieved with the ROM approach. Each look up table based controller has a ROM memory and a counter. The counter serves as an address pointer for the ROM memory. The counter is incremented when all input data is assigned to all Execute unit's input arguments.

### 3.2.1 Example

In Table 3.2, we show the ROM table sizes for a number of parameterized applications. The applications are: QR which is a matrix decomposition algorithm [43], stereo vision which is a 1-D motion estimation algorithm [44], and optical flow which is an image restoration algorithm [45]. For each application, we give for a particular processor in the network the size of the ROM table needed to program the Read and Write units local controller. The processors shown correspond to the processors with the largest ROM of a benchmarked algorithm. For each of the processors selected, we change the algorithm's parameters to observe the memory footprint. Consider the processor number $4$ of the PN representing stereo vision. This processor requires a ROM of $459,016$ bytes for a $320 \times 200$ image (i.e., $W = 320, H = 200$), and $1,941,576$ bytes for a $640 \times 400$ image (i.e., $W = 640, H = 400$). If we apply a simple run-length encoding scheme, we can compress the ROM memory to $401.320$ bytes from $459,016$ bytes for a $320 \times 200$ image. Suppose we map the node on a Virtex-II 6000 device. This device has a maximum memory of $3,648$K bits available. Hence, implementing the control table of a single processor would already consume $88$% of the available memory bits on the FPGA, which is totally unacceptable.

| | Parameters | | Uncompressed implementation (bytes) | Compressed implementation (bytes) | % of mem FPGA |
|---|---|---|---|---|---|
| | N | T | | | |
| QR (processor 4 out of 5) | 8 | 16 | 448 | 400 | 0.08 |
| | 16 | 64 | 7680 | 4160 | 0.9 |
| | 64 | 256 | 516096 | 78080 | 17.12 |
| | W | H | Uncompressed | RLE | % of mem |
| Stereo vision (processor 4 out of 5) | 320 | 200 | 459016 | 401320 | 88.00 |
| | 640 | 400 | 1941576 | 1698240 | 372.42 |
| | 1024 | 640 | 5072328 | 4437264 | 973.08 |
| | W | H | Uncompressed | RLE | % of mem |
| Optical Flow (processor 3 out of 7) | 320 | 200 | 944460 | 14850 | 3.25 |
| | 640 | 480 | 3808860 | 29850 | 6.54 |
| | 1024 | 764 | 9780540 | 47850 | 10.4 |

Table 3.2: Control ROM size for three different applications

### 3.2.2 Discusion

The ROM table controller is the best solution to achieve small clock delays for the system, as no additional computation is needed. However, this approach appears to be impractical for large iteration domains, since the ROM size grows with the volume of the domain. Therefore, these ROM tables can quickly exceed the storage capacity of most FPGAs. Hence, the ROM approach is impracticable for large iteration domains. Whenever the size of the domains is small then this approach remains feasible and very fast.

## 3.3 The Parameterized Predicate Controller

Another way to implement the controller of the virtual processor is to evaluate the linear expressions as they appear in the program of a processor of a CDFPN. Due to the class of nested-loop programs that we consider, all the iteration domains are bounded by linear expressions. These linear expressions are in fact predicate. For example, in the program listed in Listing 3.1 port domain $IPD_3$ is guarded by a number of predicates: $i \leq j$, $j \leq 4$, $2 \leq i$ and $i \leq 4$. By evaluating these predicates at run-time, the VP that implements the $IPD_3$ can select a proper channel to receive data from. A big advantage of this approach is that the control is still parameterized in the original parameters of the application. We can therefore change the parameters in FPGA, once per each application instance.

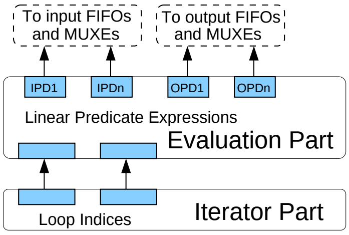


Figure 3.3: Two key components of a Parameterized Predicated Controller

As shown in Figure 3.3, a predicate controller consists of two parts : an Iterator Part and an Evaluation Part. In the *iterator part*, the predicate controller iterates over the looped domain. In the *evaluation part*, the predicate controller evaluates in parallel and at every iteration, all the linear expressions to determine the output ports or input ports to be selected. Thus, the iterator part provides the values of the iterators, $i$ and $j$, that are used in the evaluation part to evaluate the predicates $i \leq j$, $j \leq 4$, $2 \leq i$ and $i \leq 4$. A parameterized predicated controller can be derived in four steps, as described below.

**Redundancy Elimination** The first step consists in reducing the computational load by applying a common expression elimination to remove all redundant expressions present in the predicates associated to the IPDs/OPDs. This is a very effective step, as in the vast majority of the targeted applications, this optimization can eliminate the computations involved in the controller up to 80%.

**Dependency Graph Construction** In the second step, a dependency graph is constructed. This *dependency graph* is associated with the linear expressions involved in the loops and IPD/OPD predicates. A dependency graph ($G = (N, E)$) is a representation of all data dependencies between all the operations of the linear expressions. Each node ($N_i \in N$) in the graph represents a linear expression operation (i.e., arithmetic operation such as addition, multiplication, or logical operation such as logical and or logical comparator). Additionally, a node can implement a select function which selects between two inputs. The select node is used to initialize a particular operator. Directed edges ($E_j \in E$) are used to represent the data flow between the graph nodes. The graph is directed and acyclic. This graph representation of the computation within the linear expression unveils the parallelism that exists in such linear expressions. During this step computations in both the iteration and evaluation parts are combined.

**Mapping** In the third step, the dependency graph is mapped onto a parallel data-path, by associating with each operation/node in the graph its hardware equivalent. We map the data-dependence graph as a pure combinational data-path with only its output being synchronized through registers. Additionally, a feed-back loop is created to enable the updating of the state of the loop iterators (e.g., one for-loop depends on the value of a pervious for-loop).

**Bitwidth Selection** The last step of this data-path synthesis operation is to determine the bitwidth of the various operators involved in the data-path. This analysis is crucial since it allows to drastically improve both area usage and performance figures.

Since the dependency graph associated with the predicate evaluations is always acyclic (the predicate expressions only depend on the current loop indic values), all the data-path operator bitwidths can be derived from the original loop indices bitwidth, which themselves depend on the loop bounds. Thus, if $ub(i)$ is the upper bound for loop index $i$, then its bitwidth is given by $w_i = \lceil \log_2(ub(i)) \rceil$. The loop indexes and the program parameters are the source nodes of the directed graph. A source node is a node with no input edge.

Using this information, we then propagate the bitwidth constraint along the arithmetic nodes of the dependency graph, using equation 3.4, in which $in_1$ and $in_2$ represent the graph node Input Port bitwidths, and $f$ is the operation performed in the node at hand. In our case, all the arithmetic nodes implements arithmetic operation that operates only on integer unsigned numbers.

$$w(f, in_1, in_2) = \begin{cases} \max(w(in_1), w(in_2)) + 1 & \text{if} \quad f = add \\ w(in_1) + w(in_2) & \text{if} \quad f = mul \\ w(in_1) - w(in_2) + 1 & \text{if} \quad f = div \\ w(in_2) & \text{if} \quad f = mod \end{cases} \tag{3.4}$$

For the logical nodes, the bitwidth is always one as their results is a boolean value. For the select nodes, the bitwidth is $w_k = \max(w(in_1), w(in_2))$. The bitwidths of all the edges of the dependency graph are computed by solving the equations in each graph node. The graph is traversed from its source nodes to its sink nodes. A sink node is a node that has no output edge.

### 3.3.1 Example

We will now take a particular node (i.e., $V$) of the QR algorithm (see Listing 3.2) as an example and derive a parameterized predicated controller. Figure 3.4 shows the loop-iterators `k` and `j` and a number of predicates to activate the proper Input Port operation, i.e., read from a FIFO and Output Port operation, i.e., write to a FIFO. At each iteration of the loop-iterators, some predicates are evaluated to read the correct data from the correct FIFO. This is given by the `READ` part. Next the function $V$ is executed in the `EXECUTE` part. Finally data is written to the correct FIFO in the `WRITE` part.

Listing 3.2: QR algorithm in Matlab

```
for k = 1: 1: N,
    for j = 1: 1: T,
        [r(j,j),t] = V( r(j,j),x(k,j));
        for i = j+1: 1: K,
            [r(j,i),x(k,i)] = R(r(j,i),x(k,i),t);
        end
    end
end
```

```
void P2 ::main() {
 for (int j = 1 ; j <= N ; j += 1 ) {
   for (int k = 1 ; k <= T ; k += 1 ) {

       if (k-2 >= 0)                        READ
         in_0 = read(FIFO1);                IPD_1
       if (k-1 == 0)
         in_0 = read(FIFO2);                IPD_2
       if (j-2 >= 0)
         in_1 = read(FIFO3);                IPD_3
       if (j-1 == 0)
         in_1 = read(FIFO4);                IPD_4

       (out_0,out_1,out_1)=              EXECUTE
                V(in_0,in_1) ;

       if (-k+T-1 >= 0)                    WRITE
         write(out_0,FIFO1);                OPD_1
       if (-j+N-1 >= 0)
         write(out_1,FIFO5);                OPD_2
       if (k-T == 0)
         write(out_2,FIFO6);                OPD_3

   } // for k
 } // for j
```

Figure 3.4: Description of a process take from the PN representation of the QR algorithm, each **read** (resp. **write** ) operation is guarded by one or more linear expression, that depend on the loop parameters and indices

From the algorithm given in Figure 3.4, the linear expressions are extracted from the loop-iterators as follows:

```
j = 1;
j ≤ N;
j += 1;
```

```
k = 1;
k ≤ T;
k += 1;
k − 2 ≥ 0;
k − 1 == 0;
j − 2 ≥ 0;
j − 1 == 0;
−k + T − 1 ≥ 0;
−j + N − 1 ≥ 0;
k − T == 0;
```

After applying common expression elimination, a dependency graph $G$ is constructed. A node $N_i \in N, G = (N, E)$ is a linear expression operation. Each linear expression operation operates on two input operands. Exception of this rule is made of the selection nodes that have three inputs: one for control selection and two for selected data.

A linear expression operation has a single output that is either an arithmetic output or a logical output. The type of output depends on the type of the linear expression operation (i.e., either logical or arithmetical operation). The next lines show each node of the dependency graph $G$, where out($n_x$) is the output of a node $n_x \in N$.

```
n1( jt, 1, ’+’ );
n2( jt, 1, ’−’ );
n3( N, 1, ’−’ );
n4( jt, 2, ’−’ );
n5( T, 1, ’−’ );
n6( kt, 1, ’+’ );
n7( kt, 2, ’−’ );
n8( kt, 1, ’−’ );
n9( jt, out(n5), ’−’ );
n10( jt, out(n3), ’−’ );
n11( kt, 1, ’−’ );
n12( out(n11), 0, ’≥’ );
n13( out(n2), 0, ’==’ );
n14( out(n10), 0, ’==’ );
n15( out(n4), 0, ’≥’ );
n16( out(n7), 0, ’≥’ );
n17( out(n8), 0, ’==’ );
n18( out(n11), 0, ’==’ );
n19( out(n21),out(n1), 1, ’select’ );
n20( out(n12),out(n6), 1, ’select’ );
n21( out(n9), 0, ’≥’ );
```

The dependency graph of the linear expression of the program given in Figure 3.4 is constructed as given in Figure 3.5. At the top of the dependency graph are shown four ports. Two ports for the parameters N and T and two ports for the current loop-iterators $k_t$ and $j_t$. The lower two port correspond to the values of $k_{t+1}$ and $j_{t+1}$ which will be feedback as inputs to the dependency graph after each iteration execution. Each time the dependency graph is evaluated, a particular switch pattern appears at the IPD and OPD outputs driving the Read and Write units in the Virtual Processor as shown in Figure 3.1.

### 3.3.2 Discusion

As opposed to the ROM approach, the parameterized predicate controller implementation allows us to handle large parameterized iteration domains. Still, we observe two issues. The first issue relates to the complexity of the domain (size, shape) spanned by the loop-iterators and on the number of *IPDs/OPDs* involved in a node. Since all linear expressions have to be evaluated at every cycle, their evaluation requires more resources as the domain gets

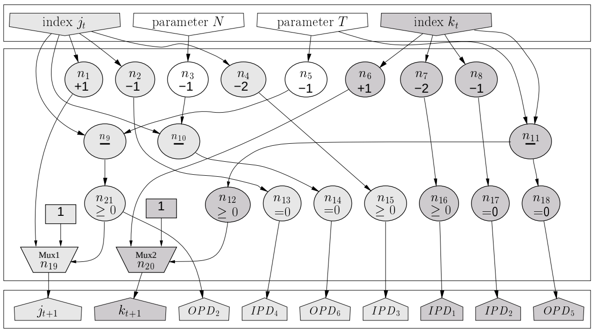


Figure 3.5: *Dependency graph* associated to the loop described in Figure 3.4. Light gray nodes correspond to operations that depends on the most inner loop index, and therefore need to be evaluated at each new iteration, dark gray nodes correspond to operations that depends on the most outer loop index, and therefore only need to be re-evaluated when the inner loop upper bound is reached.

more complex, and the number of linear expression grows. As a consequence, especially for irregularly shaped domain, the controller implementation might use a lot of area. The second issue relates to the fact that we map all predicate evaluations as pure combinational functions. As a consequence, the controller speed (i.e., the frequency at which the controller can run) might not be scalable: computational complexity (i.e., its number of *terms* in the predicate expressions) usually increases with the number of dimension of the iteration domain. The resulting controller critical path will therefore be very dependent on the number of dimension of the domain.

## 3.4 The Partitioned Parameterized Predicate Controller

In Section 3.3, predicates are evaluated at each and every iteration implying that a large amount of computation is done each iteration. However, only some predicates need to be evaluated every iteration. Such predicates are the one that have a data dependency on the most inner loop. Hence, it is possible to drastically reduce the amount of computations by restricting the computation to be done whenever is required. To do so, we take advantage of the fact that expressions that depends on the most inner loop index have to be evaluated at every cycle. Other linear expressions only need to be re-evaluated when one of their associated loop index changes value. This happens when one of the outer-loop iterators has reached its upper-bound. We can employ two techniques to partition the computation in computation that must be evaluated each cycle and computation that is not evaluated each cycle. These techniques are:

- Annotating the dependency graph derived in Section 3.3 with information when each node has to execute;
- Partitioning the dependency graph derived in Section 3.3 in two parts. One part evaluates the predicates that are critical using dedicated logic. The other part evaluates the

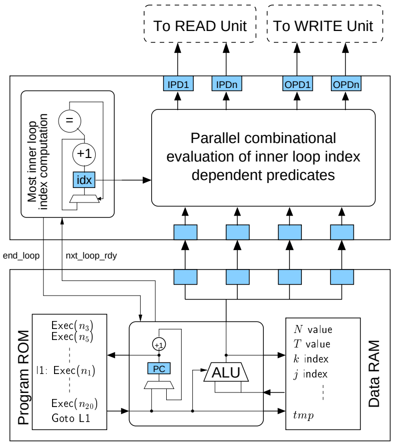


Figure 3.6: Partitioned hardware loop controller architecture

predicates using a more sequential approach like a sequencer.

The technique of annotating the dependency graph is discussed in Chapter 6. It is intended to be used for handling complex linear expression computation (e.g., with multipliers and integer division operations). This technique has the same advantages and limitations as the parameterized controller when we consider hardware resources consumption.

The second technique that partitions the dependency graph leads to a new type of controller called *partitioned parameterized predicate controller*. The partitioned parameterized predicate controller takes advantage of the data dependency between loop-iterators and predicates to get a simplified controller that uses less resources. As less expressions have to be evaluated, fewer resources are needed to realize the controller. In the partitioned parameterized predicate controller, the controller is split into two parts: a *parallel data-path*, and a *sequential controller*. The parallel data-path is similar to the one presented in Section 3.3. Its purpose is to evaluate all expressions depending on the value of the *most inner loop-iterator*. The sequential controller, on the other hand, evaluates all the expressions that depend on the outer loop iterators and parameters (including the most inner loop upper and lower bounds) and forward its results to the parallel data-path.

In Figure 3.6, the parallel data-path is given in the top part, and the sequential controller is given in the bottom part. In the sequential controller, a small sequential program is stored in a *Program ROM*. This sequential programs computes the values of the outer-loops predicates in a number of steps using some intermediate values. These intermediate values are stored in a *Data RAM*. The final values are forwarded to the parallel data-path. The parallel data-path evaluates at each cycle the parallel combinational logic, using the forwarded values. The only value that is re-computed, is the inner loop-iterator iteration. The partitioned parameterized predicate controller is generated in two steps:

**Dependency Graph Partitioning** In the first step, we apply again common expression elimination, and construct the data dependence graph as explained previously in Section 3.3. The main difference is that in this case, we will partition this graph into a set of sub-graphs (one for each loop index). Each sub-graph contains all the nodes which have as input argument the sub-graph corresponding loop index. In case a node has two distinct loop-iterators as argument, we map the node to the sub-graph that is associated with the most inner loop-iterator.

**Mapping** In the next step, we partition the computation between the parallel and sequential controller. This partitioning uses the sub-graphs obtained in the dependency graph partitioning. All the nodes of the sub-graph associated with the most inner loop index are mapped on the parallel controller. All arguments that depend on values calculated by the sequential controller are mapped on communication ports with the sequential controller. The remaining sub-graphs are mapped on the sequential controller. We sequentially schedule the operations in the remaining sub-graphs, by performing a topological sort to ensure that data dependence constraints as satisfied. Then we generate the global sequential schedule by concatenating all these local schedules in a decreasing depth order. Values that need to be communicated to the parallel controller are mapped on communication ports with the parallel controller.

### 3.4.1 Example

As an example of how we derive a partitioned parameterized predicated controller, we look again at the $V$ node of the QR algorithm as given in Figure 3.4. The first step is to obtain the partitioned dependency graph. The dependency graph shown in Figure 3.5 is therefore decomposed in three sub-graphs: $G_{param}$, $G_j$ and, $G_k$, which are respectively associated with the parameters N and T, and to the two loop indices $j$ and $k$.

Next we need to map a sub-graph onto the parallel controller or the sequential controller. Since $j$ is the inner loop iterators, we map $G_j$ onto the parallel controller, where $V(G_j) = \{n_1, n_2, n_4, n_9, n_10, n_{13}, n_{14}, n_{15}, n_{19}, n_{21}\}$. The two remaining sub-graphs are mapped onto the sequential controller. This means that the sequences of operations $V(G_{param}) = \{n_3, n_5\}$, and $V(G_k) = \{n_7, n_{16}, n_8, n_{17}, n_{11}, n_{18}, n_6, n_{12}, n_{20}\}$ are sequentialized to a program that runs on the sequential controller as shown in Figure 3.7. The **write()** instruction forwards the outer-loops predicate values to the parallel data path using dedicated ports. The **synchronize()** instruction is used to synchronize the two components of the partitioned controller. The synchronization is needed to not overwrite outer-loops predicated values by the sequential controller.

### 3.4.2 Discussion

Some problems might appear when the most inner loop domain becomes very small (only a few iterations). In such a case, the sequential machine might not be able to compute the partial results needed by the inner-loop parallel datapath fast enough, therefore slowing-down the controller. In some applications, processors does not need to fire at every cycle either because they are in a blocking read or blocking write state, or because the IP core cannot start the execution of a new operation in every cycle. Thus, the impact of the slow computation in the control sequential machine is then very unlikely to impact the overall network performance.

```
// parameters
0  k   := 1;
1  n3  := N-1;
2  write(n3,port1);
3  n5  := T-1;
4  write(n5,port2);
// k outer loop
5  n7   := (k-2);
6  n16  := n7>=0;
7  write(n16,port3);
8  n8  := (k-1);
9  n17  := n8=0
10 write(n17,port4);
11 n11  := (k-T);
12 n18  := n11=0;
13 write(n18,port5);
14 n6:= k+1;
15 n12  := n11>=0;
16 if n12
       n20:=n6;
   else
       n20:=1;
   end if;
17 k   :=n20;
// sync with datapath
18 synchronize()
19 jump 3
```

Figure 3.7: A possible schedule for the sequential controller

| | Parameters | | Non partitioned | | Partitioned | | Look-up table | |
|---|---|---|---|---|---|---|---|---|
| | $N$ | $T$ | $MHz$ | Area | $MHz$ | Area | $MHz$ | Area |
| QR factorization | 8 | 16 | 140 | 29 | 100 | 112 | 150 | 112 |
| | 16 | 64 | 133 | 68 | 85 | 133 | 80 | 1920 |
| | 64 | 256 | 121 | 89 | 74 | 163 | N.A. | 129024 |
| | $W$ | $H$ | $MHz$ | Area | $MHz$ | Area | $MHz$ | Area |
| Stereo-vision | 320 | 200 | 97 | 133 | 65 | 120 | N.A. | 114754 |
| | 640 | 400 | 100 | 148 | 74 | 123 | N.A. | 485394 |
| | 1024 | 640 | 100 | 153 | 71 | 126 | N.A. | 1268082 |
| | $W$ | $H$ | $MHz$ | Area | $MHz$ | Area | $MHz$ | Area |
| Optical-flow | 320 | 200 | 129 | 97 | 76 | 98 | N.A. | 236115 |
| | 640 | 400 | 118 | 110 | 72 | 103 | N.A. | 952215 |
| | 1024 | 640 | 126 | 113 | 75 | 106 | N.A. | 2445135 |

Table 3.3: Experimental results for partitioned and non-partitioned parameterized predicate controller

## 3.5 Conclusions

To observe the benefits of our approaches with respect to the ROM Table implementation, we used the same applications as in Table 3.2. For each of them we derived both a partitioned and a non-partitioned parameterized predicate controller, which was then mapped on a Xilinx Virtex-2 FPGA. The results, given both in terms of frequency and resource usage (in FPGA *slices*) are shown in Table 3.3. From these results, we can make the following remarks.

- The controller resource usage can vary a lot, depending on both the application and on the domain size (which influences the bitwidth of the operators in the data-path).
- It appears that the fixed area cost overhead caused by the sequential controller used in the partitioned approach is not negligible, and makes this implementation strategy only viable for very large and complex domains.
- In all cases, the partitioned controller is slower than its counterpart. This slow-down is mostly due to the sequential controller.

In general, these two strategies provide interesting results, even for very simple and small domains, and therefore suggest that the ROM table approach is not really appropriate whenever large iteration domains are involved (as it is the case for most image processing algorithms).

In this chapter we have investigated three different approaches for efficiently deriving the control part in Virtual Processors. From the experimental results, it turns out that in order to derive the optimal controller implementation, the characteristics of the application (like domain size and shape, loop nest dimension, number of function arguments or number of IPDs/OPDs) have to be taken into consideration. From the different approaches proposed in this chapter, two of them (ROM and parametric predicate) are automated in the LAURA tool.

# Chapter 4

# Communication Synthesis

In the conversion from a CDFPN to an Abstract Architecture, we already indicated that each channel of the Abstract Architecture is classified to one of four communication types. In this chapter, we investigate the synthesis issues of the four communication types and discuss the effectiveness of the realizations. We start with analyzing the communication channel types in Section 4.1. Next, we present in Section 4.2, the synthesis templates for all communication channel types found in an Abstract Architecture. We end this chapter with a case study (Section 4.3) and conclusions in Section 4.4.

## 4.1 Background

A simple instance of a CDFPN is a producer process that communicates with a consumer process over a FIFO channel. Much of the material used in this section is from [6, 7].

Consider the producer:

```
for j = 1: 1: N,
    for i = j: 1: N,
        [a(j,i)] = F_P();
    end
end
```

and the consumer

```
for k = 1: 1: N,
    for l = k: 1: N,
        [] = F_C(a(k,l));
    end
end
```

The producer is characterized by the domain

$$\mathcal{P} = \{ \begin{pmatrix} j \\ i \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} [N] \geq \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \}$$

and an *order* of the tokens that are produced (for $N = 8$):

$(1,1)$ $(1,2)$ $(1,3)$ $(1,4)$ $(1,5)$ $(1,6)$ $(1,7)$ $(1,8)$ etc.

The order of any produced token is given by a *rank* polynomial [6, 46]:

$$r_P(j,i) = -\frac{1}{2} * j^2 + (N + \frac{1}{2}) * j + i - N$$

The consumer is characterized by the domain

$$\mathcal{C} = \{ \begin{pmatrix} k \\ l \end{pmatrix} \in \mathbb{Z}^2 | \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} [N] \geq \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \}$$

and an *order* of the tokens that are consumed:

$(1,1)$ $(1,2)$ $(1,3)$ $(1,4)$ $(1,5)$ $(1,6)$ $(1,7)$ $(1,8)$ etc.

The order of any consumed token is given by the *rank* polynomial:

$$r_C(k,l) = -\frac{1}{2} * k^2 + (N + \frac{1}{2}) * k + l - N$$

The data dependency relation between the producer and consumer is given by the following mapping function $f$:

$$f \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} k \\ l \end{pmatrix}$$

such that $r_C(k,l) = (r_P \circ f)(k,l)$. The rank polynomial $r_P()$ is called the *write polynomial* and the rank polynomial $r_C()$ is called the *read polynomial*.

### 4.1.1 The Order of Producing and Consuming Tokens

Let us consider the sequence of producing and consuming tokens in more detail. In above example, the producer produces tokens in the same order as the consumer consumes them. Hence, the communication between the producer and consumer is *in-order*. In this case, a simple FIFO communication model is enough to realize the communication requirements between a producer and a consumer. However, when a consumer has to read tokens in a different order, a FIFO communication model will not do [6, 41, 46].

Consider a different consumption sequence given by the consumer:

```
for k = 1: 1: N,
    for l = 1: 1: k,
        [] = $F_C$(a(l,k));
    end
end
```

The order of the tokens that are consumed is ($N = 8$):

$(1,1)$ $(1,2)$ $(2,1)$ $(2,2)$ $(3,1)$ $(3,2)$ $(3,3)$ $(4,1)$ etc.

The mapping function and the reading polynomial are modified to deal with this new ordering of consumption.

The mapping function $f$ is:

$$f\binom{j}{i} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \binom{k}{l}$$

and

$$r_C(k,l) = -\frac{1}{2} * l^2 + (N + \frac{1}{2}) * l + k - N$$

The consumer process has to skip a number of tokens produced by the producer before the token to be consumed (e.g., to read token $(2,1)$, the consumer process needs to skip 6 tokens produced: $(1,3)$, $(1,4)$, $(1,5)$, $(1,6)$, $(1,7)$, and $(1,8)$). Thus, the FIFO communication model brakes here due to the *out-of-order* communication behavior. In such case, we use the write polynomial and the read polynomial to store and to load a token to/from a memory. This memory is called the *reorder memory* [46], and lays between the producer process and consumer process. We see, latter on in this chapter, what is the implications of such out-of-order communication for our Abstract Architecture.

### 4.1.2 The Lifetime of a Token

The order is not the only discrimination factor when we talk about the communication between a producer process and a consumer process. Usually, a producer process stores (writes) a token ether in a FIFO or in a reorder memory. The consumer loads (reads) this token from the FIFO or reorder memory, destroying (or releasing) the respective location. There can be a problem when a token is used more than once by the consumer process. We say that a token has a *multiplicity* [47] when the token has to be reused a number $W$ times.

Consider the consumer:

```
for k = 1: 1: N,
    for m = 1: 1: k,
        [] = F_C(a(k,k));
    end
end
```

The order of the tokens that are consumed is ($N = 8$):

$(1,1)$ $(2,2)$ $(2,2)$ $(3,3)$ $(3,3)$ $(3,3)$ $(4,4)$ $(4,4)$ etc.

With the mapping function $f$ defined as:

$$f\binom{j}{i} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \binom{k}{m}$$

and read polynomial as

$$r_C(k,m) = -\frac{1}{2} * k^2 + (N + \frac{1}{2}) * k + k - N$$

The tokens are consumed *in-order* from the main diagonal, multiple times by the consumer process.

Figure 4.1 graphically shows the producer iteration domain and the consumer iteration domain. The consumer process consumes *in-order* multiple times the tokens produced by the producer process on the diagonal points (white points). The arrows shows the execution schedule of the producer and consumer processes.

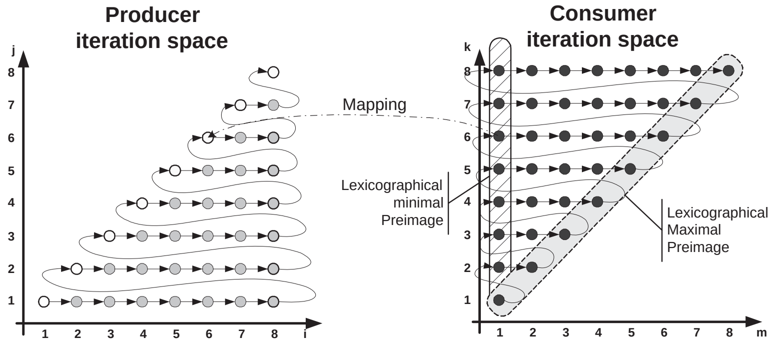


Figure 4.1: Lifetime analysis over a P/C pair with multiplicity

The multiplicity of a token is, in fact, expressing the life time of a token. As shown in [47], there are two ways to determine a memory location containing a token multiplicity larger than one can be reused:

- the *Lexicographical minimal Preimage* (LmP) indicates when a token is read from a channel for the first time;
- the *Lexicographical Maximal Preimage* (LMP) indicates when a token is read from a channel for the last time.

In Figure 4.1, we graphically show the lexicographical minimal preimage, and the lexicographical maximal preimage of the consumer iteration domain. Clearly, we can have token multiplicity also in an out-of-order communication.

### 4.1.3 Communication Types: Overview

Four types of communication channels [41] can be distinguished as illustrated in Figure 4.2. They result from the *ordering* of the iterations in the producer, and the consumer processes and the *multiplicity* of a token.

Depending on the order of consuming tokens, and the multiplicity of tokens read, a communication belongs to one of four disjoint classes: *In-Order without multiplicity* (IOM-), *In-Order with multiplicity* (IOM+), *Out-of-Order without multiplicity* (OOM-), and *Out-of-Order with multiplicity* (OOM+). In [41], it is stated that on average the following distribution can be expected over the various communication types: type IOM- (80%), IOM+ (10%), OOM- (9%), OOM+ (1%). Types IOM- and IOM+, count for 90% of the communication channels. They require a FIFO buffer to be realized. In the remaining 10% of the cases, a more complex communication structure is needed. This communication structure between a

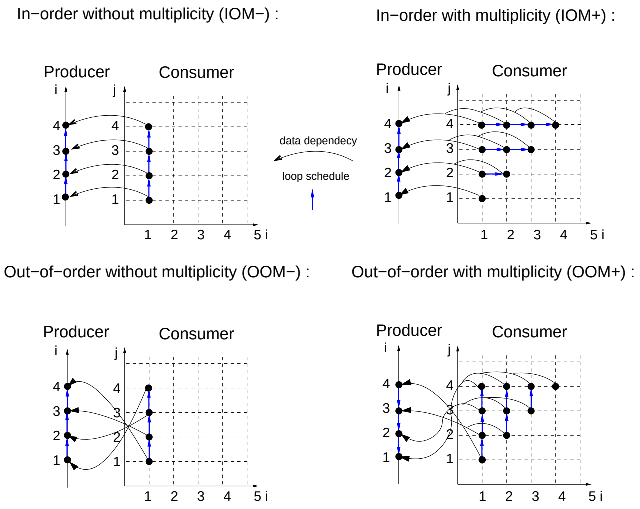

In−order without multiplicity (IOM−) :
In−order with multiplicity (IOM+) :
Out−of−order without multiplicity (OOM−) :
Out−of−order with multiplicity (OOM+) :
Producer
Consumer
i
j
data dependecy
loop schedule

Figure 4.2: The four cases of communication between Producer and Consumer

producer and consumer is extended to include additional memory to store tokens that have been produced but will be consumed latter on [46]. In this chapter, we investigate different synthesis templates that are needed to realize these types of communication in FPGAs.

## 4.2 Communication Channel Template in Abstract Architecture

To implement a CDFPN, we need to synthesize the four communication types. The topological mapping maps a process to a processor (i.e., Virtual Processor). In the *In Order* communication case (IOM-), the consumer processor reads tokens from a channel in the same order as they are written to the channel by the producer processor, it also consumes the read tokens in this order. Therefore, a FIFO buffer is all what is needed to relate produced and consumed tokens. Today's high optimized FIFO implementation buffers [48] require for each read or write primitive only a single cycle to execute. Actual FIFO buffers have finite capacity, thus both read and write primitives are blocking, i.e., they halt a processor when no data is available in a FIFO buffer or when a FIFO buffer is full. To find a finite capacity for the FIFO buffers is a problem that is addressed in Chapter 5. To find a near minimum finite capacity of the FIFO buffers is a problem that has been addressed in [49].

In the *In Order with Multiplicity* communication case (IOM+), the consumer processor consumes a token that is read from a channel more than once. Because reading from a channel is destructive, the consumer processor has to store such a token locally. In this case, the lifetime of a token needs to be taken into account. Only at the end of the lifetime of the token, the memory location, where the token is saved can be re-used. Lifetime analysis is based on the lexicographical minimal preimage (LmP), and is implemented as part of the consumer processor. The communication channel between the producer processor and consumer processor is realized using a FIFO buffer with a finite capacity. However, the FIFO buffer is modified to have sticky output, i.e., the last token read from the FIFO buffer is kept at the output port of the FIFO buffer until a new token is read. Thus, the sticky output implements the memory location that has to save a token with multiplicity. The Read unit of the Virtual Processor accommodates the LmP control. This control inhibits the reading from the FIFO buffer when a consumer iteration is out of the LmP domain, and enables the reading from the FIFO buffer otherwise. Hence, implementing the reading and releasing of tokens with multiplicity.

In the case of *Out of Order* communication cases, the producer processor and the consumer processor follow a different order of reading and writing the tokens into a channel. To obtain a correct implementation execution behavior in the out-of-order case, we need a mechanism to store and order the produced tokens for the use of the consumer processor. This mechanism relays on the usage of the *reorder memory* for temporary storage of tokens. Once stored, the tokens can be consumed in the correct order. In [50], the authors named this mechanism the Extended Linearization Model (ELM), and they implemented it as part of the consumer process. However, instead to make the reorder mechanism part of the consumer processor, we define a new channel template to implement the ELM as part of an Abstract Architecture communication channel.

### 4.2.1 The Channel Template for the Extended Linearization Model Realization

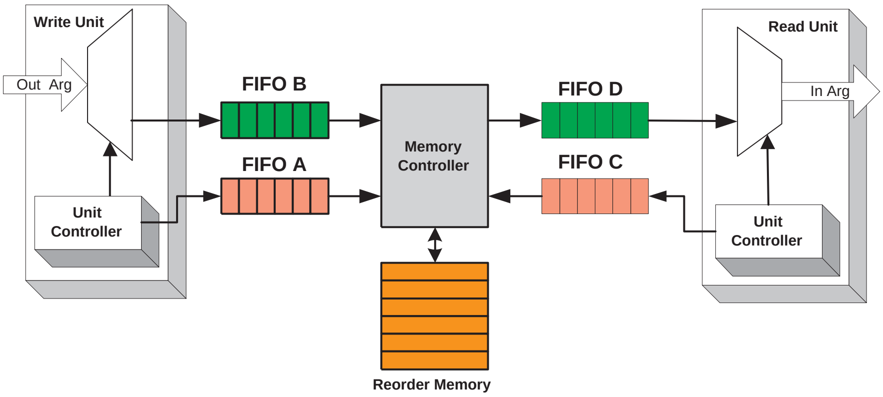


Figure 4.3: The channel template for the ELM mechanism

The main elements in the ELM are the *reorder memory* and the *memory controller*. Because the tokens can no longer be read directly from a FIFO buffer, as they arrive in the wrong order, they are delivered by the memory controller to the consumer processor. In Figure 4.3, a schematic representation is given of the ELM. It shows the Write unit of the producer processor, the Read unit of the consumer processor, the reorder memory, and the memory controller. The producer processor communicates with the memory controller via two FIFO buffers (i.e., FIFO A, and FIFO B). The consumer processor communicates with the memory controller via two FIFO buffers (i.e., FIFO D, and FIFO C). The tokens are sent using the FIFO B by the producer processor to the ELM mechanism. The unit controller of the Write unit is modified to send a write address via FIFO A. Each token send to the memory controller has associated an address send via FIFO A by the Write unit controller. The memory controller writes the token in the reorder memory at the address indicated by the token's write address.

The tokens are received using the FIFO D by the consumer processor from the ELM mechanism. The unit controller of the Read unit is modified to send a read address via FIFO C. The memory controller reads the reorder memory location indicated by the read address and writes the requested token in the FIFO D. If the token is not yet produced, the reading token from the reorder memory operation is placed on hold until the token is produced. In the case of *Out of Order with Multiplicity* communication type, the read address placed in the FIFO C is annotated with a boolean value that indicates the release command of the read reorder memory location. The boolean value is generated in function of the lifetime of a requested token. Lifetime analysis is based on the lexicographical maximal preimage (LMP), and implemented as part of the consumer's Read unit controller. The LMP control indicates to release a reorder memory location when a consumer iteration point is in the LMP domain.

In general, the capacities of the ELM mechanism FIFO buffers are arbitrarily large given the fact that the reorder memory is large enough to avoid artificial deadlocks of the network. If the reorder memory is set to its minimal size, then the FIFO B and FIFO A buffers have to be sized such that the network deadlock situations are avoided. In practice, the reorder memory capacity is large enough to avoid the network deadlock situations. This capacity is determined using the techniques described in Chapter 5. In this case, the ELM mechanism FIFO buffers capacities are set to their minimal value (i.e., a single location) to minimize resources consumption.

### 4.2.2 The Extended Linearization Model Modifications in the Read Unit and Write Unit Controllers

In the case of an ELM mechanism the Read unit and the Write unit controllers are modified to accommodate the read generation of the write address and the read address. The write address is generated by the write function, as well the read address is generated by the read function, as discussed in Section 4.1. However, these functions implementation in a FPGA is not really an option in practice, not even when multiplications are avoided by using the method of differences [51], see also Chapter 6. This is because the complexity of the write and read functions are directly related to the complexity of the Output Port Domain (OPD) and Input Port Domain (IPD) polyhedral shape (see Section 4.1). A way to overcome the problem is by replacing the OPD and IPD polyhedrons with *bounding boxes*. A bounding box is a hyper rectangle that enclose a polyhedron. Thus, the write and read functions for bounding boxes are affine functions. Finding bounding boxes is a minimization problem that we consider in Chapter 5.

The Read or Write unit controllers are modified to hold these addressing functions. Consider the Parameterized Predicated Controller discussed in Chapter 3. In Figure 4.4, we graphically show the modifications. In the *address evaluation part*, the read and write addresses are evaluated in parallel, using the methodology discussed in Chapter 6. In the *LMP evaluation part*, the predicate controller evaluates in parallel all the linear expressions of the LMP iteration domain to determine if a reorder memory location can be released or not.

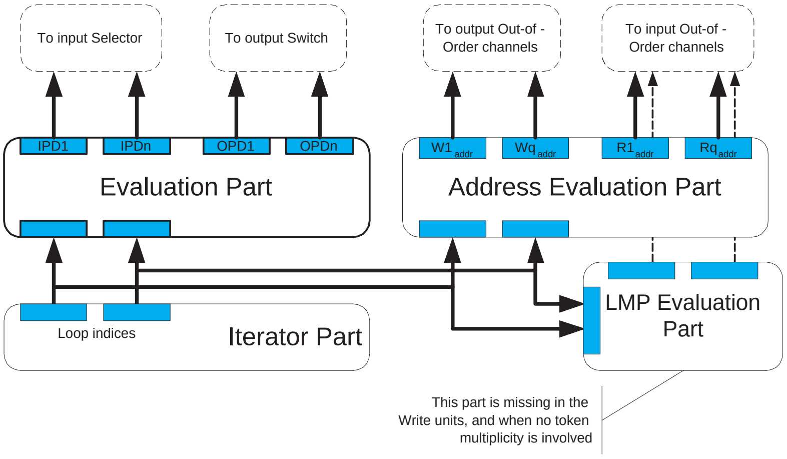


Figure 4.4: The modified Parameterized Predicated Controller

The synthesis of the LMP evaluation part is the same as for the evaluation part of the parameterized predicated controller. The LMP evaluation part is only present in the Read unit controllers when token multiplicity is involved (OOM+ communication cases).

## 4.3 FPGA Realization

The VHDL output of our methodology has been tested on a VirtexII-6000 platform from Xilinx. A FIFO channel is implemented using different types of memory, chosen at compile time. The selection is based on the FIFO buffer capacity. If less than $1024$ bits are required, we use RAM16x1D memories, otherwise we use Xilinx Block RAM (RAMB) memory blocks which are more suited to large memory bank implementations. If only a single location is required, we simply instantiate a FIFO channel that uses only a register. In case of the ELM mechanism implementation, we always use RAMB memories to realize the reorder memory. The hardware realization of a FIFO buffer is the fastest and most efficient one in terms of clock cycles per operation, as it requires one clock cycle per read or write operation. In the case of a ELM mechanism implementation, the write operation requires one to two clock cycles, depending of the memory estimation model chosen. The read operation requires three clock cycles to read one scalar token from the channel.

### 4.3.1 Example

To highlight the different characteristics of the communication channels, we looked at the QR algorithm [52] as shown in Listing 4.1. This algorithm requires 1 IOM+ and 10 IOM- channels. However for the experiment, we can also implement these channels using 1 OOM+ and 10 OOM- channels. This allows us to expose the differences in implementation and performance. In both cases, the lower bounds on the channels are determined at compile time using the procedure given in Chapter 5.

Listing 4.1: QR algorithm in Matlab

```
for k = 1: 1: K,
    for j = 1: 1: N,
        [r(j,j),t] = V( r(j,j),x(k,j));
        for i = j+1: 1: N,
            [r(j,i),x(k,i)] = R(r(j,i),x(k,i),t);
        end
    end
end
```

| | | |
|---|---|---|
| Software Implementation | Memory locations | 154 |
| | Memory Size | 4928 bits |
| Reorder Implementation | Clock Cycles | 258 |
| | RAMB | 11 |
| | Memory Size | 180224 bits |
| | Slices | 1771 |
| | Frequency | 100 Mhz |
| FIFO Implementation | Clock Cycles | 128 |
| | RAM16x1D | 416 |
| | Memory Size | 6656 bits |
| | Slices | 1006 |
| | Frequency | 100 Mhz |

Table 4.1: Experimental results for various hardware channels.

Table 4.1 shows the memory requirements, and FPGA performance figures for the QR algorithm. We also considered a software implementation of the test case to underline the difference in memory requirements for different implementation cases. The out-of-order communication type channels are more inefficient than FIFO channels in terms FPGA memory usage. The Reorder Memory is realized using only Xilinx Block RAMs (RAMB). For the 11 channels, 11 RAMB memory blocks are allocated, which represents 180224 bits of the FPGA memory. For each ELM mechanism implementation, we need to allocate at least a RAMB block which can accommodate minimum 512 tokens of 32 bit. Hence, it is difficult to use the memory block efficiently; there can be quite some spill.

On the other hand, for a FIFO channel, we can select between Xilinx Block RAMs or smaller RAM16x1D memories, depending on the required memory size. In the QR implementation case, we use 416 RAM16x1D memory blocks for the 10 IOM- channels and 1 IOM+ channel. If we choose to use only out-of-order communication type channels to implement the QR algorithm we need almost twice the number of clock cycles (i.e., 258 cycles) compared with a FIFO implementation (i.e., 128 cycles). The rather complex protocol of reading/writing to an ELM mechanism implementation has a heavy influence on the overall performances of an algorithm. The number of slices used to implement the QR algorithm using only FIFOs is 1006, when using only ELM mechanism implementation it is 1771. The out-of-order communication type channel requires twice as many slices due to the address generators. Both the execution speed and number of slices used show the importance of proper selection and implementation of a channel communication type.

## 4.4 Conclusions

Four point-to-point communication types are distinguished as part of our methodology to decouple the computation from communication [53]. This decoupling allows the IP cores (the computation part) and the interconnect (the communication part) to be designed separately. In this chapter, we have shown that for each communication type derived by the COMPAAN methodology, we can derive efficient communication architectures in FPGAs. Two communication types use FIFO buffer implementations and two communication types use an ELM mechanism implementation. From the case study, we shown that a FIFO implementation is the most optimal implementation of a communication channel from any point of view (throughput, hardware resources, memory usage).

Our methodology focuses on data-flow algorithms, having communication at the level of scalars (e.g., bytes or words). The communication topology of the CDFPN is static, and derived at compile time. To realize the inter-processor communication, we use a point-to-point communication mechanism. Employing busses and/or complex Networks-on-Chips (NoCs) [54, 55] for the communication is not feasible due to the delays in the routing process and the usage of large packets instead of scalars in the communication protocol.

Chapter 5

# Memory Bound Estimation

When synthesizing a channel, whether it is an in-order communication type channel or an out-of-order communication type channel, we need to determine a finite channel capacity, that is large enough to avoid network deadlocks, and small enough to avoid over capacity. In this chapter, we investigate different strategies to determine the size of all the channels in an Abstract Architecture. We first introduce the problem in Section 5.1. Next in Section 5.2, we provide the mathematical background needed to analyze the memory requirements in an Abstract Architecture. Depending on the channel communication type, we can estimate the storage capacity in two ways. The first approach is presented in Section 5.3, and is applicable only to in-order communication type channels. The second approach is presented in Section 5.5, and is applicable to all communication type channels.

## 5.1 Maximum Size of FIFO channels

A process network is specified in terms of partial orderings between a producer and a consumer, i.e., the Producer/Consumer pair. To find a bound on a PN channel requires, however, a total order on the execution of the processes in the network. Many different total orders exist, leading to different trade-offs in evaluation speed and memory requirement. A total order for a CDFPN can be obtained by scheduling or by doing a run-time evaluation of a process network [56]. Nevertheless, we would like to avoid both methods. The scheduling interferes with the notion of distributed control, and the run-time evaluation is not a compile time analysis that can be performed as part of our methodology. Also, both the run-time and schedule approach cannot handle the parameterized nature of the CDFPNs we derive. Instead, we use compile time allocation schemes. We can do so due to the SANLP characteristic of input programs that are used to derive the CDFPNs. In [49], the authors determine a lower dimension for FIFO buffers of their similar process networks.

Consider, the program listed in Listing 5.1 is a simple sequential SANLP in which the function $St1()$ produces a variable $a(j)$ that is consumed by the function $St2()$, forming a P/C pair (as defined in Definition 2.1). A classical memory allocation procedure [57, 58] would reserve a memory of size $N$ to store results $a(j)$ of $St1()$. We observe here that the variable $a(j)$ is allocated as a vector, and its allocated memory cells are reused during the execution of the algorithm. Hence, the type of program code listed in Listing 5.1 that reuses memory locations is referred as *sequential code*.

Listing 5.1: SANLP example: sequential code

```
for i = 1 : 1 : N,
   for j = i : 1 : N,
      a(j) = St1();
   end

   for j = i : 1 : N,
      ... = St2(a(j));
   end
end
```

The program listed in Listing 5.2. The memory allocation procedure [57, 58] would reserve a memory of size $N \times N$ to store results $a(i,j)$ of $St1()$. We observe here that the variable $a(i,j)$ is allocated as a matrix, and its allocated memory cells are not reused during the execution of the algorithm. Hence, the type of program code listed in Listing 5.2 that do not reuse memory locations is referred as *single assignment code*. Please note that the type of single assignment code used by us [59] is slightly different than the general accepted concept of a single assignment code. The difference consist that only the data flow related variables (e.g., $a(i,j)$) are using single assignment memory allocation. The control flow related variables (e.g., $i, j$) are not converted to a single assignment memory allocation.

Listing 5.2: SANLP example: single assignment code

```
for i = 1 : 1 : N,
   for j = i : 1 : N,
      a(i,j) = St1();
   end

   for j = i : 1 : N,
      ... = St2(a(i,j));
   end
end
```

However, the Output Port Domain for the function $St1()$ has only $\frac{1}{2} * N * (N + 1)$ elements and so has the input-port domain for the function $St2()$. Clearly, no more than $\frac{1}{2} * N * (N + 1)$ tokens have to be exchanged between functions $St1()$ and $St2()$. As a consequence, the FIFO size of the communication channel between the two processes in the corresponding CDFPN can safely be bounded to $\frac{1}{2} * N * (N + 1)$. Thus, if an Output Port Domain contains $M$ points, then the FIFO size of the channel to which this domain connects need never be larger than $M$. Counting the number of points in a domain $\mathcal{D}$ can be done by means of the Ehrhart pseudo-polynomial count procedure [42]. Ehrhart counting pseudo-polynomial expressions may turn out to be complicated when the underlying domain is not trivial and/or communication of the tokens is out-of-order. To overcome this problem, domains may be approximated by enclosing them in simple bounding boxes [60].

The programs listed in Listing 5.1 and Listing 5.2 are functional equivalent. As we have seen, the only difference is the way in that results of function $St1()$ are stored in memory.

In the first case, we require $N$ memory locations, and in the second case, we require $N \times N$ memory locations. Moreover, by taking advantage of the SANLP structure of the given programs, we have seen that the results of function $St1()$ can be stored in $\frac{1}{2} * N * (N + 1)$ memory locations no matter what allocation type we use. In the following sections, we analyze each of these memory allocation methods, and what are the effects of these methods over our Abstract Architecture.

## 5.2 Volumes and Lexical Addressing Functions

A channel is defined as a pair of ports, an Input Port and an Output Port, both being lexically ordered domains. Given a domain $\mathcal{D}$, we now define two functions as follows:

**Definition 5.1** $volume(\mathcal{D}) : \mathcal{D} \rightarrow \mathbb{Z}$, is the number of distinct integral points a polyhedral domain $\mathcal{D}$.

**Definition 5.2** $lex(\mathcal{D}) : \mathbb{Z}^n \rightarrow \mathbb{Z}$, is a lexical ordering of the integral points in a polyhedral domain $\mathcal{D}$.

In a P/C pair, we can randomly access the communication memory using a read address (at the consumer side) and a write address (at the producer side). We can generate those addresses using a *write* and *read* functions associated to P/C pair communication memory. Those functions are obtained by only tacking into account the producer domain (i.e., OPD).

**Definition 5.3** The write function, $write : \mathbb{Z}^n \rightarrow \mathbb{Z}$ is defined as:

$$write(j) = \{lex(j) \mid j \in \mathcal{D}_P\} \tag{5.1}$$

where $\mathcal{D}_P$ is the Output Port Domain of a channel, $j$ is an iteration point that belongs to OPD, and $lex()$ perviously defined.

**Definition 5.4** The read function, $read : \mathbb{Z}^n \rightarrow \mathbb{Z}$ is defined as:

$$read(i) = \{(write \circ f)(i) \mid i \in \mathcal{D}_C\} \tag{5.2}$$

where $f()$ is the mapping function of the P/C pair and $\mathcal{D}_C$ is the consumer domain (i.e., IPD), and $write()$ perviously defined.

For the in-order channels $write()$ and the corresponding $read()$ are identical and defined as simple counters. The $write()$ and $read()$ functions are not required for the in-order communication type channels as the communication memory is linearized to a simple FIFO. However, for the out-of-order $write()$ is not a simple counter, and its counter part function $read()$ is more complicated (see Chapter 4). The out-of-order channels are addressing directly the communication memory, and we require simple functions to do so. We discuss how to obtain affine functions for these memory addressing functions in Section 5.5.

## 5.3 In-Order Channel Upper-bound Memory Estimation

In the COMPAAN Data Flow Process Networks, the FIFOs communication channels are unbounded. Actual implementation FIFO sizes are undecidable. In an Abstract Architecture, on the other hand, channel FIFOs can be given finite upper-bounds which guarantee that the network will never deadlock. Before dealing with the problem of deciding on the size of a particular channel FIFO, we take the view that a producer process writes results to a local memory that can be accessed by the corresponding consumer process. The question, then, is what the upper bound of the size of the local memory could be. The answer depends on whether the process is a single assignment code (SAC) or a non-single assignment sequential program (SEQ).

Listing 5.3: Non-single assignment example code

```
for i = 1 : 1 : 10,
    for j = 1 : 1 : i,
        if (i+j >= 10)
             a(j) = St1();
             ... = St2(a(j));
        end
    end
end
```

Listing 5.4: Single assignment example code

```
for i = 1 : 1 : 10,
    for j = 1 : 1 : i,
        if (i+j >= 10)
             a(i,j) = St1();
             ... = St2(a(i,j));
        end
    end
end
```

Consider the non-single assignment sequential program listed in Listing 5.3, and the corresponding single assignment code program listed in Listing 5.4. In both programs, the function $St1()$ is defined on the domain $\mathcal{D}$ given by:

$$\mathcal{D} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \ \wedge \ 1 \leq j \leq i \ \wedge \ i+j \geq 10\}$$

The domain is the polygon show in Figure 5.1. The program listed in Listing 5.3 has an output variable indexing function:

$$j = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

The program listed in Listing 5.3 has an output variable indexing function:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

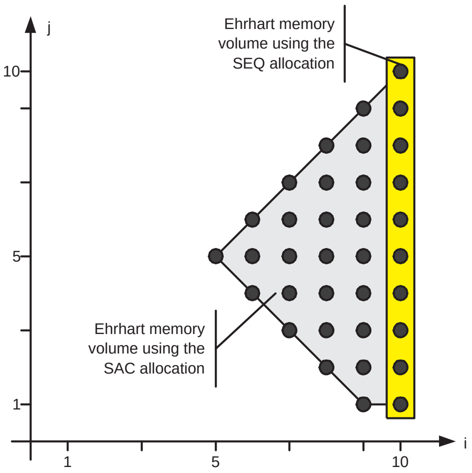

j
i
10
5
1
1
5
10
Ehrhart memory volume using the SEQ allocation
Ehrhart memory volume using the SAC allocation

Figure 5.1: The graphical representation of the $\mathcal{D}$ domain

Thus, for the Output Port of the output variable $a$ of the function $St1()$ can have two different Output Port Domains. The two Output Port Domains are:

$$\mathcal{OPD}_{SEQ} = \{j \in \mathbb{Z} \mid 1 \leq i \leq 10\}$$

and

$$\mathcal{OPD}_{SAC} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 10 \ \wedge \ 1 \leq j \leq i \ \wedge \ i + j \geq 10\} \equiv \mathcal{D}$$

The $\mathcal{OPD}_{SEQ}$ graphical representation is also shown in Figure 5.1.

To find the volume of these domains, we can use the $Polyhedron\_Enumerate()$ function of the PolyLib library [61]. This function uses Ehrhart polynomials [42] to count all the integral points of a polyhedral domain $\mathcal{D}$. Thus, $volume(\mathcal{OPD}_{SEQ}) = 10$ and $volume(\mathcal{OPD}_{SAC}) = 35$. These volumes represents different memory sizes allocated to a communication channel. The size of the memory allocated to $\mathcal{OPD}_{SEQ}$ is a safe upper-bound provided that the domain is scheduled in the order given by the sequential program. On the other hand, the size of the memory allocated to $\mathcal{OPD}_{SAC}$ is an upper-bound irrespective of the domain schedule chosen.

## 5.4 Self-loops Channel Memory Estimation

In general, we cannot determine the size of the FIFOs in a CDFPN as the processes are only partially ordered. To determine the size of each FIFO would require to globally schedule the network. A self-loop is, however, a special case as this kind of communication channel starts, and ends on the same processor and the writing and reading to/from this channel respects the internal schedule of the processor. Thus, the IPD and the OPD of the channel are subsets of the same iteration domain. We exploit this property to determine the size of the self-loop channels.

Listing 5.5: SANLP with a self-loop

```
for i = 2 : 1 : 10,
   [a(i)] = So();
end

for i = 1 : 1 : 5,
   for j = i : 1 : 5,
      a(i+j) = F1(a(i+j));
   end
end
```

Consider the SANLP example shown in Listing 3.1. It is reproduced here in Listing 5.5. In this example we show also the initialization of the $a$-array in the first loop. The processor that implements statement $So$ is feeding the $F1$ processor via two FIFOs that connect to the $IPD_1$ and $IPD_2$ domain of the processor. The domains of the processor $F1$ are as follows:

$$IPD_1 = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 1 \ \wedge \ 1 \le j \le 5\} \tag{5.3}$$

$$IPD_2 = \{(i,j) \in \mathbb{Z}^2 \mid 2 \le i \le 5 \ \wedge \ 5 \le j \le 5\} \tag{5.4}$$

$$IPD_3 = \{(i,j) \in \mathbb{Z}^2 \mid 2 \le i \le 4 \ \wedge \ i \le j \le 4\} \tag{5.5}$$

$$OPD = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 5 \ \wedge \ i \le j \le 5 \ \wedge \ 2 \le j - i\} \tag{5.6}$$

$$ND = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 5 \ \wedge \ i \le j \le 5\} \tag{5.7}$$

The Abstract Architecture network of the example given in Listing 5.5 is depicted in Figure 5.2.

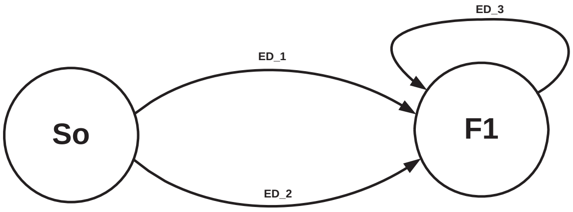


Figure 5.2: The Abstract Architecture network of the example given in Listing 5.5

The self-loop appears due to a data dependency between the input argument of function $F1$ and its output argument. Thus, all data produced by the $F1$'s $OPD$ is consumed by its $IPD_3$ (e.g., $(i,j) = (1,3) \rightarrow (2,2), (1,4) \rightarrow (2,3)$, etc.). In this example, the self-loop FIFO requires a size of three locations as the data produced in iteration $(1,3)$ is consumed later in iteration $(2,2)$. Hence, all data produced in $OPD$ between these two points has to be stored. The size of this storage gives us the size of the self-loop channel and it is defined as the maximum amount of tokens stored in the FIFO or, equivalently, the life-time of a token written to the channel FIFO.

To detect the FIFO size of a self-loop, we use the Ehrhart theory to count how many integral points are contained in a polytope. The *rank* function [46] ranks an iteration point $x$ that belongs to an iteration domain $\mathcal{D}$. The rank of the iteration point $x$ is the number of iteration points that, where executed before iteration point $x$. For simplicity, we consider only a rank function with one validity domain [6].

A first step is to find how many tokens are produced before the first token is consumed. We call this period the *run-in* period. The first token consumed from a self-loop channel is in the first iteration in the IPD. This iteration is the lexicographic minimum point of the IPD domain ($min_{\mathbf{lex}}\{IPD\}$). The point $x \in OPD$ that is lexically the first point produced before the first point in IPD satisfies the following parametric integer linear program (PIP):

$$\begin{array}{rl} \textbf{subject to:} & x \in OPD \\ & x \prec min_{\mathbf{lex}}\{IPD\} \\ \textbf{objective:} & max_{\mathbf{lex}}\{x\}, \end{array}$$

Let $x_{wr}$ be the unique solution to this program. Then the number of tokens produced in a run-in period $Ntokens_{run-in}$ is given by:

$$Ntokens_{run-in} = rank_{OPD}(x_{wr}), \tag{5.8}$$

where $rank_{OPD}()$ is the rank function for the self-loop OPD.

After the run-in period comes the steady period in which tokens are produced as well as consumed. The steady period extends from $min_{\mathbf{lex}}\{IPD\}$ and $max_{\mathbf{lex}}\{OPD\}$. The number of tokens that has been produced in this period is given by:

$$Ntokens_{produced} = rank_{OPD}(max_{\mathbf{lex}}\{OPD\}) - rank_{OPD}(x_{wr}) \tag{5.9}$$

and the number of tokens consumed is given by:

$$Ntokens_{consumed} = rank_{IPD}(x_{rd}). \tag{5.10}$$

where $rank_{IPD}()$ is the rank function of the IPD, and $x_r d$ is the unique solution to the following PIP:

$$\begin{array}{rl} \textbf{subject to:} & x \in IPD \\ & x \prec max_{\mathbf{lex}}\{OPD\} \\ \textbf{objective:} & max_{\mathbf{lex}}\{x\}, \end{array}$$

Using Equation 5.9 and Equation 5.10, we can compute the absolute number of tokens produced in the steady period as:

$$Ntokens_{steady} = rank_{OPD}(max_{\mathbf{lex}}\{OPD\}) - rank_{OPD}(x_{wr}) - rank_{IPD}(x_{rd}) \tag{5.11}$$

If $Ntokens_{steady}$ is negative then we consume more than we produce in the steady period. If $Ntokens_{steady}$ is positive then we produce more than we consume. Otherwise, we neither consume or produce in the steady period.

A last period is the *run-out* period. In this period remaining tokens are consumed without any production of tokens. Clearly, this period does not have any impact on the determination of the self-loop FIFO size. Thus, the self-loop FIFO size is equal with the total number of tokens produced in the run-in and steady periods and not consumed yet. The FIFO size is given by:

$$FIFOsize = max(Ntokens_{run-in}, Ntokens_{run-in} + Ntokens_{steady}) \tag{5.12}$$

The above result FIFO size is valid when the domains IPD and OPD are polyhedral domains.

### Self-loop Example

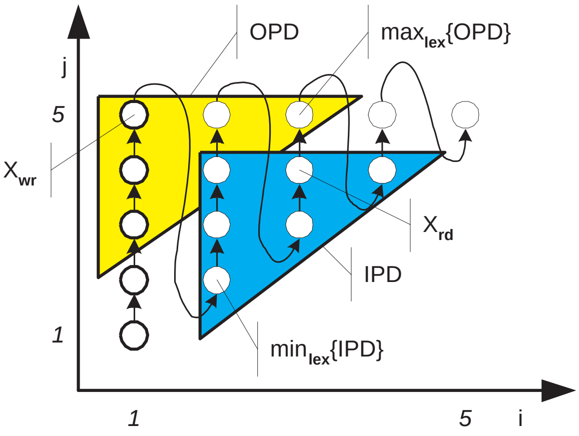


Figure 5.3: The iteration domain of statement $F1$, the IPD, and OPD of the self-loop FIFO. The arrows indicate the lexicographical schedule

Consider the program given in Listing 5.5. The self-loop FIFO OPD and IPD are depicted in Figure 5.3. $min_{\mathbf{lex}}\{IPD\}$ is given by the point $(2, 2)$. $max_{\mathbf{lex}}\{OPD\}$ is given by the point $(3, 5)$. Thus, solving the PIP problems we get $x_{wr} = (1, 5)$ and $x_{rd} = (3, 4)$. The ranks are as follows:

$$\begin{aligned} rank_{OPD}(max_{\mathbf{lex}}\{OPD\}) &= 6 \\ rank_{OPD}(x_{wr}) &= 3 \\ rank_{IPD}(x_{rd}) &= 5 \end{aligned}$$

So, the results of Equation 5.8 and Equation 5.11 are:

$$\begin{aligned} Ntokens_{run-in} &= 3 \\ Ntokens_{steady} &= -2 \end{aligned}$$

Thus, the self-loop FIFO size is given by Equation 5.12 that is $FIFOsize = 3$.

## 5.5 Memory Estimation using Bounding Boxes

There are cases when the n-D communication memory between a producer and a consumer cannot be linearized to a FIFO [41, 46, 62]. In these cases (i.e., the out-of-order cases), the consumer needs to address the n-D communication memory using the *read* function (Definition 5.4), and the producer using the *write* function (Definition 5.3), respectively. When linearized, the n-D memory is converted to a FIFO structure, and, hence, the read and write functions are simple counters. However, for the out-of-order cases, the n-D memory is converted to a 1-D memory that requires special read and write functions to access it. In [46], the authors proposed to use the *rank()* function as basis for the read and write functions (i.e., the $lex()$ function, see Definition 5.2). The $rank()$ function is defined as a summation of pseudo-polynomial expressions. The complexity of these pseudo-polynomial expressions depends on the shape of the iteration domain that is ranked. Thus, the complexity of the synthesized architecture depends on the complexity of these pseudo-polynomial expressions. We want to limit as much as possible the complexity of the synthesized architecture by limiting the complexity of the read and write functions, and implicit the complexity of the $lex()$ functions.

The complexity of the lexical function $lex()$ can be reduced by enclosing an arbitrary iteration domain $\mathcal{D}$ in a *bounding box*. The bounding box is a hyper-rectangular shape that can be converted to a one dimensional array using classical linearization methods given in [57, 58]. Hence, the read and write functions that address this linearized bounding box are affine functions. The complexity of the lexical function depends now only on the number of dimensions of the bounding box that includes the iteration domain, and not on the shape of the iteration domain. In this section, we show how we enclose an arbitrary iteration domain in a bounding box that is a rectangular domain.

### 5.5.1 Background

Let a rectangular domains be defined as:

$$\mathcal{D} = \{x \in \mathbb{Z}^n \mid l \leq x \leq u\} \tag{5.13}$$

where $x = (x_1, x_2, \ldots, x_n)$, $l = (l_1, l_2, \ldots, l_n)$, and $u = (u_1, u_2, \ldots, u_n)$.

We can define $lex$ and $volume$ functions for a rectangular domain in terms of a vector $w$ whose elements $w_i$ are defined recursively in terms of $l_i$ and $u_i$:

$$w_i = \begin{cases} 1 & \text{when } i = 1; \\ w_{i-1}(u_{i-1} - l_{i-1} + 1) & \text{when } 1 < i \leq n + 1. \end{cases} \tag{5.14}$$

The $lex$ and $volume$ functions are defined in terms of $w$:

$$lex(v) = (w_1\ w_2 \ldots\ w_n) \begin{pmatrix} v_1 - l_1 \\ v_2 - l_2 \\ \vdots \\ v_n - l_n \end{pmatrix} \tag{5.15}$$

$$volume(\mathcal{D}) = w_{n+1} \tag{5.16}$$

where $n$ is the dimension of $\mathcal{D}$, $v = (v_1, v_2, \ldots, v_n)$ is the iteration vector of iteration domain $\mathcal{D}$, and the vector $w$ is known as the doping vector in compiler terminology [57].

Table 5.1 shows rectangular domains of dimensions $1$, $2$, and $3$ with their associated $lex$ and $volume$ functions.

| | |
|---|---|
| 1-D domain | $lex(i) = i - L_i$<br>$volume(i) = (U_i - L_i + 1)$ |
| 2-D domain | $lex(i,j) = (U_j - L_j + 1)(i - L_i) + j - L_j$<br>$volume(i,j) = (U_j - L_j + 1)(U_i - L_i + 1)$ |
| 3-D domain | $lex(i,j,k) = (U_k - L_k + 1)(U_j - L_j + 1)(i - L_i) +$<br>$(U_k - L_k + 1)(j - L_j) + k - L_k$<br>$volume(i,j,k) = (U_k - L_k + 1)(U_j - L_j + 1)(U_i - L_i + 1)$ |

Table 5.1: Lexical and volume functions for rectangular domains

Figure 5.4 shows a bounding box for the OPD of the statement $ST1$ in the program listed in Listing 5.4. The OPD iteration domain $\mathcal{D}$ is inside of its bounding box $\mathcal{D}_{\mathcal{BB}}$. In our approach, we allocate $volume(\mathcal{D}_{BB})$ memory for the OPD domain $\mathcal{D}$. The wasted memory is proportional to $Volume(\mathcal{D}_{BB}) - Volume(\mathcal{D})$.

For the bounding box $\mathcal{D}_{BB}$, we have:

$$\begin{aligned} lex(i,j) &= (10 - 1 + 1)(i - 5) + j - 1 \\ volume(i,j) &= (10 - 1 + 1)(10 - 5 + 1) \end{aligned}$$

In Listing 5.4 the upper and lower bounds of the iteration vectors are not explicitly given. Thus, we need a procedure to precisely determine these bounds to minimize the memory waste. In the next section, we present a methodology to determine a bounding box for any given iteration domain. Much of the material used in this section is from [63].

### 5.5.2 Deriving the Bounding Boxes

A *bounding box* is the smallest rectangular domain $\mathcal{D}_{BB}$ which contains $\mathcal{D}$. As we have seen in Equation 5.15, the lexical function depends on the boundaries $l$ and $u$. The vectors $l$ and $u$ of a bounding box can be computed for any given finite iteration domain $\mathcal{D}$. An iteration domain can be defined in function of its $k$ extremal vertices such as:

$$\mathcal{D} = \{x \in \mathbb{Z}^n \mid \begin{pmatrix} \lambda x \\ \lambda \end{pmatrix} = R\mu, \lambda > 0, \mu \geq 0\}$$

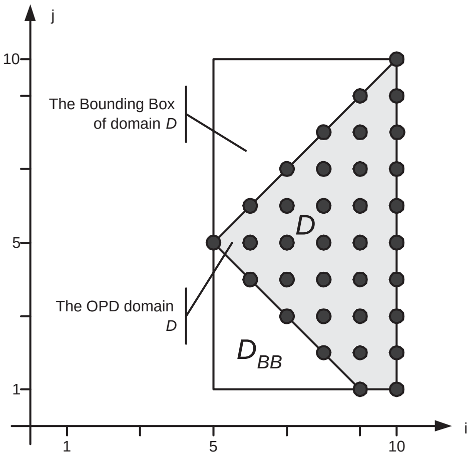


Figure 5.4: The bounding box of the OPD of statement St1 in the program listed in Listing 5.4

where $R$ is a constant integer matrix, $x$ and $\mu$ are rational vectors, and $\lambda$ is an integer scalar such that $\lambda x$ is an integer vector.

The columns of the $R$ matrix represent the $k$ extremal vertices. To bound a finite iteration domain, it is sufficient to bound its extremal vertices. The constants $l_i$ and $u_i$ are defined as minima and maxima, respectively, of the $k$ values $r_{ij}$'s in the $i^{th}$ row of $R$:

$$\begin{aligned} l_i &= min(r_{i1}, r_{i2}, \ldots, r_{ik}) \\ u_i &= max(r_{i1}, r_{i2}, \ldots, r_{ik}) \end{aligned}$$

Consider an extremal vertex $r_m = (r_{1m}, r_{2m}, \ldots, rnm)$ be the unique solution of the following PIP problem:

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{D} \\ \textbf{objective:} \quad & min_{\mathbf{lex}}\{x\}, \end{aligned}$$

then $l_i = r_{1m}$, where $i$ is the scanning dimension of the polyhedron given by the most outer loop.

Consequently, the $u_i$ constant of the $i^{th}$ dimension is given by the unique solution of the following PIP problem

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{D} \\ \textbf{objective:} \quad & max_{\mathbf{lex}}\{x\}, \end{aligned}$$

where the extreme vertex $r_M = (r_{1M}, r_{2M}, \ldots, r_{nM})$ is the unique solution, and $u_i = r_{1M}$

The PIP problems depicted above determine only two extreme vertices. These vertices determines only two constants $l_i$ and $u_i$, where $i$ is the most outer loop. Thus, we need to determine the remanning $2*(n-1)$ extreme vertices of the remanning $n-1$ dimensions. The remanning extreme vertices can be determined using the same PIP problems on a modified iteration domain $\mathcal{D}' = f(\mathcal{D})$. The modification consists in permutating the iteration domain dimensions. This permutation is know in compiler terminology [57] as loop interchange. Hence,

$$\{v = (i_1, i_2, \ldots, i_n), v \in \mathcal{D}\} \to \{v' = (i_n, i_1, \ldots, i_{n-1}), v' \in \mathcal{D}'\}$$

where $v$ and $v'$ are the iteration vectors of $\mathcal{D}$ and $\mathcal{D}'$, respectively.

A loop interchange operation is done using an unimodular transformation on an iteration domain such that all the for loop indexes are at least once in the position of the most outer loop. An unimodular transformation is defined as:

$$\exists T \ s.t. \ det(T) = \pm 1 \wedge \mathcal{D}' = \mathcal{D} \times T.$$

The unimodular transformation that do loop interchange is defined as:

$$T = \begin{bmatrix} 0 & 1 \\ I_{n-1} & 0 \end{bmatrix}$$

where $I_{n-1}$ is the identity matrix.

We can define a number of modified domains $\mathcal{D}'$ as:

$$\mathcal{D}'_i = \begin{cases} \mathcal{D}, & \text{when } i = 1; \\ \mathcal{D}'_{i-1} \times T, & \text{when } 1 < i \leq n. \end{cases}$$

All the necessary extreme vertices for the bounding box are computed as:

$$r^i_M = max_{\mathbf{lex}}\{\mathcal{D}'_i\} \tag{5.17}$$

$$r^i_m = min_{\mathbf{lex}}\{\mathcal{D}'_i\} \tag{5.18}$$

where $r^i_M = (r_{1M}, r_{2M}, \ldots, r_{nM})$ and $r^i_m = (r_{1m}, r_{2m}, \ldots, r_{nm})$ are extreme vertices of domain $\mathcal{D}$.

Hence, the constants $l_i$ and $u_i$ are:

$$u_i = r^i_{1M} \tag{5.19}$$

$$l_i = r^i_{1m} \tag{5.20}$$

The bounding box $\mathcal{D}_{BB}$ is defined in terms of $l$ and $u$.

$$\mathcal{D}_{BB} = \{x \in \mathbb{Z}^n, l \leq x \leq u\} \tag{5.21}$$

### Bounding Box Example

For the example shown in Figure 5.4, we have:

$$\mathcal{D} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \le i \le 10 \ \wedge \ 1 \le j \le i \ \wedge \ i+j \ge 10\}$$

The matrix transformation $T$ is:

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The converted domain $\mathcal{D}'$ is:

$$\mathcal{D}' = \{(j,i) \in \mathbb{Z}^2 \mid 1 \le j \le 10 \ \wedge \ 1 \le i \le j \ \wedge \ i+j \ge 10\}$$

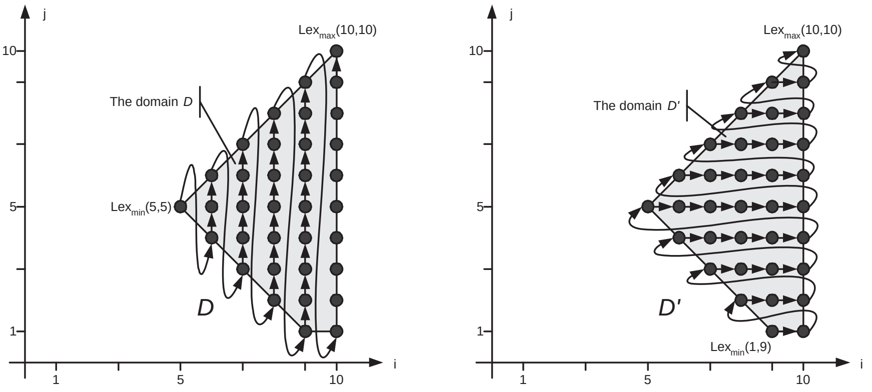


Figure 5.5: Polyhedral representation of the domains $\mathcal{D}'$ and $\mathcal{D}$. The arrows indicate the scanning order.

The purpose of the transformation $T$ is to change the way how we scan the initial iteration domain $\mathcal{D}$ to determine the extreme vertices of the iteration domain. This transformation is shown in Figure 5.5. The PIP problems gives us the unique solutions:

$$\begin{aligned} r_m^1 &= (\mathbf{5}, 5), \\ r_M^1 &= (\mathbf{10}, 10) \end{aligned}$$

for the domain $\mathcal{D}$. And,

$$\begin{aligned} r_m^2 &= (\mathbf{1}, 9), \\ r_M^2 &= (\mathbf{10}, 10) \end{aligned}$$

for the domain $\mathcal{D}'$. Thus,

$$\begin{aligned} l_i &= 5 \\ u_i &= 10 \\ l_j &= 1 \\ u_j &= 10 \end{aligned}$$

The bounding box is:

$$\mathcal{D}_{BB} = \{(i, j) \in \mathbb{Z}^2 \mid 5 \leq i \leq 10 \ \wedge \ 1 \leq j \leq 10\}$$

## 5.6 Conclusions

Due to the particularities of our CDFPN networks, we can derive at compile time an upper bound memory for our Abstract Architecture. In the case of self-loop FIFOs, the memory estimated is the lower bound memory estimation. Two memory allocation schemes are used to derive the memory type that we use to estimate an upper bound memory for a channel. The SAC allocation scheme results in the fastest execution of a CDFPN as the network can run with maximum parallelism. The SEQ allocation scheme restricts the network parallelism to a more sequential execution scheme, as explicit checks need to be performed on the validity of data. The SEQ allocation scheme can be tuned by the user, and changing the total amount of memory required by the Abstract Architecture. The tuning is realized through using source to source transformations on an input program. Such transformations are the one that increase data locality in a sequential application, i.e., loop tiling [64] and loop fusion [65].

# Chapter 6

# Expression Synthesis

A CDFPN process is mapped to a *virtual processor* that consists of a Read, an Execute, and a Write unit as explained in Chapter 2. The Read and Write units execute a control program that determines at each firing of the Execute unit from where data needs to be read and to where data needs to be written. The Execute unit embeds an IP core that implements the core function in the process. The IP Core executes a firing in a particular execution period that depends on the core's critical path. This period becomes an important design constraint for our processor, as the Read and Write units need to perform the next read and write operations in less than this period. Only then do the Read and Write unit not obstruct the data flow through the Execute unit. In Figure 6.1, we show an IP Core that executes in $10ns$. As a consequence, the Read and Write units need to prepare for the next firing in less than this $10ns$.

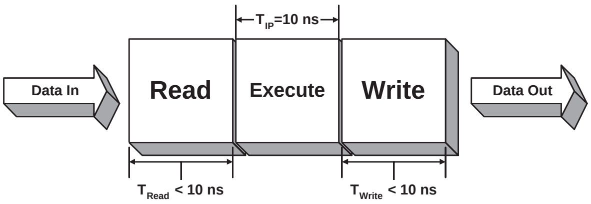


Figure 6.1: Running an IP core efficiently in a LAURA Processor

The control programs in the Read and Write units are expressed in terms of parameterized *polytopes*, IPDs andd OPDs. The parameters are static, that is, values can be set at compile time. However, an implementation is also parameterized so that the controller does not have to re-designed when parameter values change [66, 67], see Chapter 3. The IPDs and OPDs are repeatedly evaluated at run-time. If a particular iteration is within the space defined by the polytope, it means that data needs to be read or written. A polytope is defined by a set of affine and pseudo-affine relations. An example of a control program defined as a polytope is given in Figure 6.2. In it, the variables $M$ and $N$ are parameters, while $i$, $j$, and $k$ are iterators.

$$Control\ Program = \begin{cases} M-3 & \geq 0 \\ N-2 & \geq 0 \\ -j+3*M & \geq 0 \\ j-M & \geq 0 \\ 2*i-1 & \geq 0 \\ i+2 & \geq 0 \\ -k+N-1 & \geq 0 \\ k-1 & \geq 0 \\ -i+2*DIV(i+1,2)-1 & \geq 0 \\ 2*DIV(M+1,2)-M-1 & \geq 0 \end{cases}$$

Figure 6.2: Example of a set of relations that define a polytope.

To check wether an iteration point ($i$, $j$, $k$) is enclosed by the polytope relations in the control program $\mathcal{P}$ (given in Figure 6.2) or not, we have to evaluate all the conditions contained by the control program. In software, we implement this evaluation as a cascade of if-statements. The speed of evaluation of this cascade is low, as the evaluation proceeds in a sequential manner. Evaluating the polytopes in hardware, however, can be done much faster, as all expressions can be evaluated in parallel. Nevertheless, in many cases the control program cannot be evaluated in less than $T_{read}$ ($10ns$ in Figure 6.1). In this chapter, we introduce the Expression Compiler that implements a number of techniques to reduce the evaluation time of polytope expressions. As a consequence, we can evaluate expressions in the Read and Write units faster than the execution period $T_{IP}$ of the IP Core function.

We start with related work in Section 6.1. Section 6.2 deals with our approach to convert expressions efficiently to hardware in two steps. In Section 6.3 and Section 6.4, we present the techniques to simplify expressions. The predicated single assignment form is introduced in Section 6.5. Experimental results are given in Section 6.6, and we conclude this chapter in Section 6.7.

## 6.1 Related Work

A CDFPN network generated by the COMPAAN compiler may be simulated using software simulators in which all expressions that define a polytope are evaluated in a sequential order. A hardware implementation for only a very limited set of expressions (no multiplications and no pseudo linear operators) was proposed in [68]. To allow for an efficient compilation of expressions to hardware, we investigated a more flexible approach based on two observations. First, expressions can be evaluated in parallel. Second, even with a parallel evaluation of expressions, the total evaluation time can take longer than the evaluation time of an IP block embedded in our network. This is due to the complexity certain operations such as multiplication and integer division. The elimination of division and modulo operations from a sequential program has been discussed in [69, 70]. The PICO project [71] employs similar techniques to avoid MOD and DIV operations. Our target is to generate a custom implementation, and hence, additional issues have to be taken into account such as mapping, critical path delay, and the footprint of the implementation.

## 6.2 The Approach

For a fast and efficient implementation of the control program given in Figure 6.2, we simplify the given inequalities into expressions that use only additions, look up tables (LUT), and shifts. These new expressions can be executed in a shorter time than the evaluation time needed by the IP core embedded in the Execute unit. Moreover, the simplified expressions are represented in a data-structure that allows for further manipulation. We can obtain an even more optimal implementation in terms of speed and area by using, for example, pipeline, retiming and multiplexor reduction techniques. For that reason, we have broken down the transformation of expressions in two steps as shown in Figure 6.3. In the first step, the input expressions are simplified using high-level optimizations that are platform independent (i.e. High-Level Optimizer). In the second step, the expressions are manipulated to obtain better performance by taking advantage of mid and low-level optimizations that are platform dependent (i.e. Low-Level Optimizer).

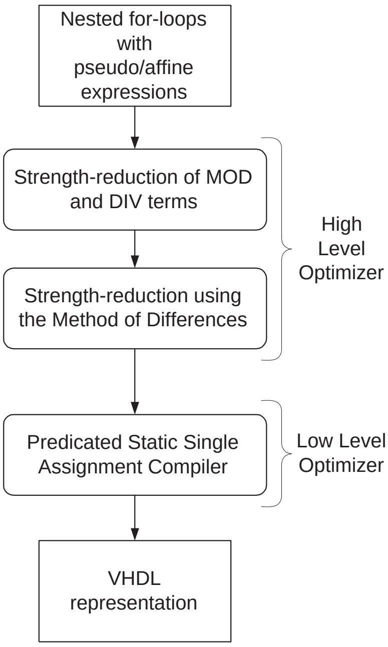

Nested for-loops with pseudo/affine expressions
Strength-reduction of MOD and DIV terms
Strength-reduction using the Method of Differences
High Level Optimizer
Predicated Static Single Assignment Compiler
Low Level Optimizer
VHDL representation

Figure 6.3: The Expression Compiler Flow

### 6.2.1 High-Level Optimizer

The High-Level Optimizer implements high-level and platform independent optimizations. It converts an expression in two steps to a simplified structure that allows for further optimization. These are:

**DIV and MOD strength reduction operations** are used to reduce the strength of expressions to simpler expressions. Also, expressions that contain DIV terms are converted to MOD terms as they are simpler to implement.

**Method of Differences** is used to replace multiplications by additions.

### 6.2.2 Low-Level Optimizer

The Low-Level Optimizer is a Predicated Static Single Assignment compiler performing mid and low-level optimizations, such as constant propagation, dead-code elimination, and retiming aiming at better performance. The Predicated Static Single Assignment Compiler compiles the input to a *Predicated Static Single Assignment* (PSSA) [72] code for a particular target architecture. We rely on PSSA to take advantage of today's research in PSSA compilation techniques [72]. The PSSA form is suitable for optimizations intended for both micro-processor [73] architectures and reconfigurable architectures [74]. Reconfigurable architectures in FPGAs are our architectures of choice. For an FPGA platform target, the desired operations are: the reduction of the number of variables used, multiplexer optimizations, bit-width, and LUT synthesis for non linear terms. Finally, the resulting PSSA is mapped onto a hardware description language like VHDL or Verilog by associating a hardware equivalent to each operation of the PSSA.

## 6.3 Simplification

A polynomial specification consist of a set of (pseudo-)affine expressions. Terms like MOD and DIV in pseudo-affine expressions require special compilation techniques for FPGA implementation. The COMPAAN compiler typically generates polytopes that contain DIV terms. The $b * DIV(a, b)$ form is the most frequently occurring form of a DIV operation. See, for example, the expressions in Figure 6.2. However, the implementation of a DIV term is more involved than the implementation of a MOD term, Therefore, we want to avoid DIV terms as much as possible. Next, an equivalence transformation that converts a DIV term to a MOD term is given. This transformation has been introduced in [75].

**Equivalence 6.1**
A DIV term of the form $b * DIV(a, b)$ is equivalent to a MOD term of the form $a - MOD(a, b)$. □

An observation is that Equivalence 6.1 can easily be extended to handle numbers that are a multiple of $b$. In the cases when we cannot rely on the equivalence 6.1, employ modern software compiler techniques to reduce the cost of integer division [76–78]. These techniques are based on so-called scaled reciprocals. In general, the techniques transform an integer division into a multiplication with a constant and a shifts [78].

The bit-width result of a DIV(a,b) term is $\lceil \log_2(a) \rceil - \lceil \log_2(b) \rceil + 1$, while the bit-width result of a MOD(a,b) term is $\lceil \log_2(b) \rceil$. Usually, we may assume that $b << a$, and we can say that the number of bits needed to represent the MOD term is less then the number of bits needed to represent a DIV term. This reinforces our goal to work with MOD rather than DIV terms. To further on optimize the MOD terms we can strength reduce them [79] as follows:

**Equivalence 6.2**

$$\begin{aligned} MOD(a*x+b*y,d) &= MOD(MOD(a,d)*x+MOD(b,d)*y,d) \\ MOD(x+y,d) &= MOD(MOD(x,d)+MOD(y,d),d) \\ MOD(x*y,d) &= MOD(MOD(x,d)*MOD(y,d),d) \end{aligned}$$

□

where x and y are variables, and a, b, and d are constants.

A special case is when the divider is a power of two constant. In that case, we can apply Equivalence 6.3.

**Equivalence 6.3**

If the divider of the MOD operation is positive power of two number, then the modulo expression can be converted to a bitwise AND operation:

$$MOD(x,\ 2^n) = x\ AND\ (2^n-1)$$

where $(2^n-1)$ is a string of $n$ ones. □

After applying Equivalence 6.2, we obtain a MOD operation with a known range of $r = [0,(d-1)], r \in \mathbb{N}$. For small dividers, this MOD operation can be implemented using a Look Up Table (LUT) as described in Optimization 6.4.

**Equivalence 6.4**

Given a modulo expression in a loop of the following form:

```
for j = L to U,
   exp = MOD(f(j),divider);
end
```

Where $f(j)$ is an affine function defined as $f(j) = f_0 + c*j$, $f_0$, $c$, and $divider$ are constants. Then the loop can be transformed to,

```
for j = L to U,
     if j = L then
          exp = MOD(f_0 + c*L, divider);
     else
          exp = MOD(exp + c, divider);
     end if
end
```

□

*Observation:* If $L$ is a constant, then value $exp$ on the TRUE branch of the if statement is a constant and can be computed at compile time. Otherwise, Equivalence 6.4 is applied recursively to further reduce the multiplication operation to additions. Equivalence 6.4 compiles a MOD term into operations that make use of LUTs. These LUTs have always maximum table length equal to $2 * divider - 1$ after applying Equivalence 6.2. The bit-width is equal to $\lceil log_2(divider - 1) \rceil$. Hence, a LUT implementation for small values of $divider$ is very well achievable.

```
for i = 0 to 2*d-1 {
     lut[i] = MOD(i,d);
}
return lut;
```

Figure 6.4: The LUT generator algorithm

To generate the content of the LUT, we make use of the algorithm given in Figure 6.4. Suppose we want to have the LUT for the MOD term $MOD(i + 2, 3)$. We need a LUT of 5 positions that is filled with $\{0, 1, 2, 0, 1\}$ and is addressed using the simple expression $i+2$. If the divider in a MOD operation becomes large, the LUT approach is no longer possible as the content doesn't fit efficiently in a FPGA slice. In that case, we make use of Equivalence 6.5, which replaces the LUT with a counter.

**Equivalence 6.5**

Let $exp_{k+1} = MOD(exp_k + c, d)$ be a MOD operation in a nested for loop. If $exp_k$ and $c$ are less than $d$, then the operation can be written as:

```
for j = L to U,
   a = exp + c;
   if a >= d then
        a = a - d;
   else
        a = a + d;
   end if
   exp = a;
end
```

□

### 6.3.1 Example

As an example of the DIV and MOD strength reduction operations, we apply Equivalence 6.1 on the last two expressions in Figure 6.2 that contain DIV terms. The result we get is given in Figure 6.5. The two complex expressions with the DIV terms have been converted to expressions that consists of a single MOD term that can be implemented in a single FPGA basic logic element (i.e., a slice). Since the dividers of both MOD terms are the same (i.e., divider = 2), only a single slice is needed in hardware to implement the original complex expressions. We could even further optimize the expressions by using Equivalence 6.3

$$\left\{\begin{array}{ll} -i+2*DIV(i+1,2)-1 & \geq 0 \\ 2*DIV(M+1,2)-M-1 & \geq 0 \end{array}\right. \Rightarrow \left\{\begin{array}{ll} MOD(i+1,2) & =0 \\ MOD(M+1,2) & =0 \end{array}\right. \Rightarrow LUT$$

Figure 6.5: Complex expressions converted to a single LUT using DIV and MOD strength reduction operations

## 6.4 Method of Differences (MoD)

In the Read and the Write units, a lexical schedule is executed. This schedule is captured by for loops. The for-loops define iteration points for which the polytope needs to be evaluated. Because of the for-loops, the expressions of a control program are evaluated repeatedly. This repetitive behavior can be exploited to simplify the evaluation of our expressions by converting all multiplication operations into repetitive addition operations. This is an important step, as a multiplication operation takes more FPGA resources and time compared to an addition.

The technique that exploits the repetitive behavior is called the *Method of Differences* (MoD) [51], as it is based on using differences of the terms of an expression to calculate the next value. Although the method of differences can be applied to a polynomial of any degree, we are dealing with pseudo-affine expressions and thus polynomials of degree one. Terms such as MOD and DIV still have to be evaluated at run-time.

Listing 6.1: Simple example

```
for (int a = 5; a<=10; a++) {
   expr = 3*a + 1;
}
```

Consider the for-loop in the simple example listed in Listing 6.1. Clearly $expr(a+1) = expr(a)+3$, $expr(5) = 16$. A MoD version of the simple example from Listing 6.1 is given in Listing 6.2.

Listing 6.2: A MoD version of the program given in Listing 6.1

```
for (int a = 5; a <= 10; a++) {
    if (a == 5) {
       b = 15
    } else {
       b = b + 3
    }
    expr = b + 1;
}
```

## 6.5 Predicated Static Single Assignment

The *Predicated Static Single Assignment* (PSSA) [72] form is suitable for mid and low-level optimizations be it for micro-processors [73] or for reconfigurable platform such as

FPGAs [74]. We are primarily interested in optimizations for a FPGA platform. The *Static Single Assignment* (SSA) [80, 81] form requires that every variable within a computation is assigned a value only once, thereby explicitly expressing the data-dependency between operations. The SSA form helps us in eliminating the anti-dependencies (Write-After-Read) and output dependencies (Write-after-Write) unveiling the hidden parallelism in the input code. The Static Single Assignment form is almost equivalent to a Dependency Graph (DG), which is a very suitable form for hardware implementations. For example, variables for intermediate results correspond to nothing more than wires that are required anyway to perform the computation. Additionally, a number of compiler optimizations are used in conjunction with SSA to optimize the output circuitry. Such optimizations that are using SSA form are:

**Dead Code Elimination** removes the code in the source code of a program that is never used;

**Constant Propagation** is the process of simplifying constant expressions at compile time;

**Sparse Conditional Constant Propagation** simultaneously removes dead code and propagates constants throughout a program;

**Partial Redundancy Elimination** eliminates expressions that are redundant on some but not necessarily all paths through a program.

By extending SSA with predication, every statement in the original computation is tagged with a *guard* that controls whether or not a statement is actually executed. Advanced techniques [74, 82] can be applied on a PSSA to optimize its output for the FPGA platform. The purpose of predication is to completely eliminate control flow statements (if-then-else,loops). Using this technique, we practically convert a control flow to a data flow. Converting SSA to a PSSA form is straitforward. For example, consider the example given in Listing 6.2. The SSA code is shown in Listing 6.3 using the Shimple format [83, 84].

The major challenge in Listing 6.3 is the synthesis of the $\phi$ nodes. In our SSA forms, we can distinguish between $\phi$ nodes accordingly to their behavior. Hence, we get two types of such nodes. The first type deals with the initialization of working variables inside of a for-loop (e.g., the $\phi$ node of the $b\_1$ variable). We call these nodes *reset-*$\phi$ nodes. The second type of $\phi$ node deals with selection of the right variable according to a condition statement in the SSA code (e.g., the $\phi$ node of the $b\_4$ variable). We call these nodes *selection-*$\phi$ nodes.

To synthesize expressions, we use the architectural structure of a parameterized predicated structure as explained in Chapter 3. Hence, we have an iterator part and an evaluation part. When we synthesize expressions, the iterator part deals with the for-loop statements. We use the for-loop statements information, however, we do not synthesize it as it is already synthesized in the context of the parameterized predicated controller. Thus, we discard from SSA form all the information related to the for-loop iterators and their updates. E.g. in Listing 6.3, the initialization of variables $a\_1$ and $a\_2$ are discarded. All the references to the $a\_1$ variable points now to an input variable generated by the iterator part (i.e. $a$). At this end, we can rewrite the SSA code into a PSSA code. For the example listed in 6.3, we get the PSSA code as given in Listing 6.4. Please note that in our case all the conditions statements are static (i.e., depends only on the iterator part). This is a consequence of the fact that the compiled expressions are part of the static control associated with a static CDFPN network. This is handy when we convert the $\phi$ functions to their if-statements equivalent (multiplexors).

Listing 6.3: SSA code of the example listed in 6.2

```
{
          b = 10;
(0)       a = 5;

      label0:
          b_1 = φ(b #0, b_4 #3);
          a_1 = φ(a #0, a_2 #3);
          if a_1 > 10 goto label3;

          if a_1 != 5 goto label1;

          b_2 = 15;
(1)       goto label2;

      label1:
(2)       b_3 = b_1 + 3;

      label2:
          b_4 = φ(b_2 #1, b_3 #2);
          expr = b_4 + 1;
          a_2 = a_1 + 1;
(3)       goto label0;

      label3:
          return;
}
```

Listing 6.4: PSSA code of the example listed in 6.2

```
{
    b = 15 : if (true);                        //initial value

    b_1 = (a == 5) ? b : b_4    : if (true);   //reset initialization
    b_2 = 15                    : if (true);   //when the iterator is reloaded
    b_3 = b_1 + 3               : if (true);
    b_4 = (a == 5) ? b_2 : b_3  : if (true);   //when the iterator is reloaded
    expr = b_4 + 1              : if (true);
}
```

The PSSA code is optimized further on by simplifying the expressions (i.e., constant propagation). The PSSA predicates are modified such that the guarded expression is executed only when one of its terms is affected by one of the for-loop iterators. This behavior is very similar with the sensibility list construction from VHDL language. For our example, the code is presented in Listing 6.5.

Listing 6.5: The PSSA code with a sensibility list

```
{
    a_event = true             : if (true);
    b_1 = (a == 5) ? 15 : b_4  : if (a_event); //reset initialization
    b_3 = b_1 + 3              : if (a_event);
    b_4 = (a == 5) ? 15 : b_3  : if (a_event); //when the iterator is reloaded
    expr = b_4 + 1             : if (true);
}
```

The PSSA form is efficiently used for low level optimizations such as bit-width optimization, minimization of the number of multiplexors, and micro pipelining. This analysis is very useful as it drastically improves both the area usage, and the performance. A data-path operating on 5 bit integers is smaller and faster than an operation on a 16 bit integer. Since the variables in an expression depend only on the loop indices, the bit-width of all operators can be derived from the bit-width of the original loop indices. The loop indices depend on the upper and lower loop bounds. Suppose that $U_i$ is the upper bound for loop index $i$, then its bit-width is given by $w_i = \lceil \log_2(U_i) \rceil$. Using this information and the fact that all the operations in a PSSA tree are additions, we can propagate the bit-width constraint along the DG structure of the PSSA. Using Equation 6.1, in which variables $in_1$ and $in_2$ have a particular bit width, we calculate the required bitwith for the result of the addition.

$$w(in_1, in_2) = max(w(in_1), w(in_2)) + 1 \tag{6.1}$$

For the final step, i.e., the synthesis of the PSSA form, we generate the circuit using only multiplexors, add/substraction operations, LUTs, and registers. By definition, all the if-statements corresponding to selection-$\phi$ nodes that are created by the expression synthesis step only check if an iterator is in its lower bound iteration point or not (see Listing 6.2). Thus, they can be synthesized as multiplexors. On the other hand, the reset-$\phi$ nodes are used to initialize the corresponding variable registers. If the architecture does not support register initialization on reset, we can use multiplexors as well. In our architecture, the iterator part of the parameterized controller can generate boolean signals that are asserted when an iterator reaches its upper bound. We take advantage of these signals and convert all the PSSA selection-$\phi$ nodes to accommodate this new condition. Hence, we re-time the entire PSSA to accommodate only upper bounds conditionals instead of lower bounds if statements. They boolean signals are called *end_loop_x*, where x is a nested for-loop iterator name. The synthesis of the VHDL code is straitforward. We connect the update of an iterator to the clock of the system and then we generate the FPGA implementation using a software engineering method called visitor [85]. In our methodology, a loop iterator is updated once per clock cycle. The VHDL code of the example shown in Listing 6.2 is given in Listing 6.6.

Listing 6.6: VHDL code of the example listed in 6.2

```vhdl
process (clk, rst)
  variable b_1_v : INTEGER range 0 to 1023;

 begin
  if rst='1' then
    b_1 := 15
  elsif rising_edge(clk) then

    if end_loop_a then
      b_1_v := 15;
    else
      b_1_v := b_1 + 3;
    end if;
    b_1 <= b_1_v;
  end if;
 end
```

## 6.6 Examples of Implementations of Expressions

In this section we show typical results obtained with our Expression Compiler, for two examples. One example (Example 1) concerns only the method of differences. The other example (Example 2) additionally uses the DIV and MOD strength reduction operations. All the experiments have been implemented except for the synthesis of the for-loops. We used the Symplify 7.2 tool for synthesis and the Xilinx ISE 6.2 tool for the mapping on a Xilinx xc2v40 platform.

### 6.6.1 Example 1

This example shows what the expression compiler does with the MOD expression $MOD(5*i,3)$. The input code with the expression is shown in Listing 6.7. First, the expression compiler applies DIV and MOD strength reduction operations to simplify the expression. Here, Equivalence 6.4 is called to analyze the *True* branch of the if-statement. It holds the initialization value for the MOD expression. Because the lower bound of the $i$ for-loop is constant, $f_0 + c * L$ is equal to 10 and the initialization of the MOD expression is $MOD(10,3) = 1$. Next, the expression compiler analyzes the *False* branch that holds the update part of the original MOD expression, i.e., $exp_{new} = MOD(exp_{old} + 5, 3)$. This part is strength reduced further on by Equivalence 6.2 as shown in Listing 6.8. Using the LUT generator presented in Figure 6.4, the content of the LUT table is initialized to $\{0,1,2,0,1\}$. The PSSA code is shown in Listing 6.9 and the equivalent VHDL code in Listing 6.10. The resulting RTL view of the PSSA compilation with our technique is shown in Figure 6.6.

Listing 6.7: Code example with a modulo-statement

```
for (i = 2; i<=10; i++) {
    modT = MOD(5*i,3);
}
```

Listing 6.8: Strength reduced modulo-statement

```
for (i = 2; i<=10; i++) {
 if (i== 2) {
    modT = 1; // MOD(10,3)
 } else {
    modT = mod3LUT(exp + 2);
 }
}
```

Listing 6.9: PSSA representation of example 6.7

```
{
    i_event = true                              : if (true);

    mod0_i_0 = (i == 2) ? 1 : mod0_i            : if (i_event);     //Reset condition

    mod0_i_1 = mod0_i_0 + 2                     : if (i_event);
    mod0_i = (i == 2) ? 1 : mod(mod0_i_1,3)     : if (i_event);

    modT = mod0_i                               : if (true);
}
```

The FPGA implementation requires only one slice to implement the MOD and its critical path is $1.5ns$ (app. $650Mhz$). The implementation results depend on the size of the look up table that has to be compiled in hardware, but for small values of the divider of the MOD, it

Listing 6.10: VHDL code of the example listed in 6.7

```
 constant mod3LUT: mod3LUT_lut := mod3LUT_lut'(
    0 => int2slv (0, mod3LUT_width+1),
    1 => int2slv (1, mod3LUT_width+1),
    2 => int2slv (2, mod3LUT_width+1),
    3 => int2slv (0, mod3LUT_width+1),
    4 => int2slv (1, mod3LUT_width+1));

process (clk, rst)
  variable modT_v : INTEGER range 0 to 1023;
  variable mod0_v : INTEGER range 0 to 1023;
  variable mod0_i_v :  INTEGER range 0 to 4;
 begin
  if rst='1' then
    mod0_i_v := 1;
    mod0_i <= mod0_i_v;
  elsif rising_edge(clk) then
    if end_loop_i then
      mod0_i_v := 1;
    else
      mod0_i_v := slv2int(mod3LUT(mod0_i+2), mod3LUT_width+1);
    end if;
    mod0_i <= mod0_i_v;
  end if;
end process;
modT <= mod0_i;
```

is suitable to be compiled using the LUT approach. This example shows that expressions that seem very complex can be implemented very efficient in today FPGAs.

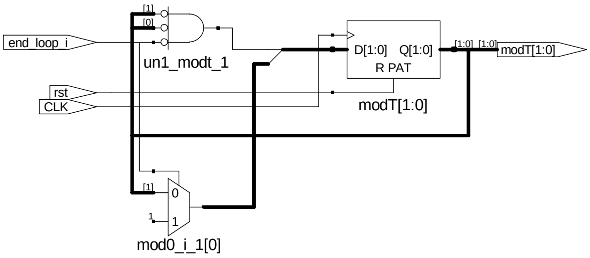


Figure 6.6: The RTL view of example 6.7

### 6.6.2 Example 2

This example shows how the expression compiler deals with the expression $81 * i + 9 * j + k - 102$ that appears in the body of for-loop nest of depth 3 (see Listing 6.11). We want to map this expression to hardware using the Expression Compiler, as the expression contains multiplications. Applying the DIV and MOD strength reduction operations will not help in this case, as pseudo-affine terms are not involved.

The PSSA code that we get for this example is given in Listing 6.13. It shows the same for-loops as in the original code with the body modified. This PSSA code is equivalent to

Listing 6.11: Code example containing non-trivial linear expression

```
for (i = 2; i<=10; i++) {
 for (j = 1; j<=i-1; j++) {
  for (k = i+1; k<=11; k++) {

   scan = 81*i + 9*j + k - 102;

  }// end k
 }// end j
}// end i
```

Listing 6.12: The intermediate code implementing the lower bound of k-loop

```
for (i = 2; i<=10; i++) {
 for (j = 1; j<=i-1; j++) {
    k3 = i + 1;
 }// end j
}// end i
```

Listing 6.13: Strength reduced code of the example 6.11

```
1   for (i = 2; i<=10; i++) {
2       if (i == 2) {
3          k3_i = 2;
4       } else {
5          k3_i = k_01 + 1;
6       }
7       k3 = k3_i + 1;
8       if (i == 2) {
9          scan_i = 162;
10      } else {
11         scan_i = scan_i + 81;
12      }
13      for (j = 1; j<=i-1; j++) {
14          if (j == 1) {
15             scan_j = 9;
16          } else {
17             scan_j = scan_j +  9;
18          }
19          for (k = i+1; k<=11; k++) {
20              if (k == 3) {
21                 scan_k = 3;
22              } else {
23                 if (k == i+1) {
24                    scan_k= k3;
25                 } else  {
26                    scan_k = scan_k + 1;
27                 }
28              }
29              scan = scan_k + scan_j + scan_i -
                       102;
30          }// end k
31      }// end j
32  }// end i
```

the code given in Listing 6.11, however all the expressions are expressed in additions only; all multiplications are removed. The expression compiler also changed the names of the variables to make the conversion to the PSSA code easier.

The expression is pure affine, and is compiled as discussed in the beginning of this chapter. Only the linear term that contains the $k$ variable is more complex than others due to the fact that the lower bound of the $k$-for-loop is an affine expression by its own. The upper bounds of the loops that are expressed as affine terms are ignored in this step. This is due to the definition of the MoD method in which upper bounds do not play any role in strength reducing of an expression.

First, the term $81 * i$ is compiled. The initial value is equal to the initialization value of $i = 2$ multiplied with its first derivative $81$.The initialization subtree is shown in Listing 6.13, lines 8 and 10. The update of this term is shown in lines 10 to 12. The entire subgraph (lines 8-12) is placed in the scope of the $i$-for-loop such that the update is done only when the update of the $i$ variable is done. Next, the $j$ term is compiled (lines 14–18), in scope of the $j$ for-loop. Finally, the $k$ term is compiled. This compilation is a bit more complex because the lower bound of the $k$ for-loop is a linear expression on its own (e.g., $i + 1$). Therefore, the expression is compiled recursively following the same steps as described above. The intermediate input code for the compiler is shown in Listing 6.12. The compilation of the new expression *k3* lies between lines 2 and 7. After that, the value of *k3* is taken and used in the initialization of the last linear term of the *scan* expression, see lines 20 to 27. We can remark that the tree corresponding to the *k3* is placed in the right place (e.g. in the scope of the $i$-for-loop). The last step consists in adding all the intermediate results in the final expression at line 29.

Listing 6.14: PSSA representation of example 6.11

```
{
    i_event = end_loop_j                    : if(true);
    j_event = end_loop_k                    : if(true);
    k_event = true                          : if(true);

    scan_i_0 = (i == 2) ? 162 : scan_i      : if (i_event);   //Reset condition
    k3_i_0 = (i == 2) ? 2 : k3_i            : if (i_event);   //Reset condition
    scan_j_0 = ( j == 1) ? 9 : scan_j       : if (j_event);   //Reset condition
    scan_k_0 = (k == i+1) ? k3 : scan_k     : if (k_event);   //Reset condition

    scan_i = (i == 2) ? 162 : scan_i_0 + 81   : if (i_event);
    k3_i   = (i == 2) ? 2 : k3_i_0 + 1        : if (i_event);
    scan_j = ( j == 1) ? 9 : scan_j_0 + 9     : if (j_event);
    k3 = k3_i + 1                             : if (j_event);
    scan_k_1 = (k == i+1) ? k3 : scan_k_0 + 1 : if (k_event);
    scan_k = (k == 3) ? 3 : scan_k_1          : if (k_event);

    scan = scan_i + scan_j + scan_k - 102     : if (true);
}
```

After strength reducing the input code, we convert it to its PSSA form as shown in Listing 6.14. The PSSA compiler generates the VHDL description as shown in Listing 6.15. The hardware implementation that corresponds to the output VHDL file is shown in Figure 6.7.

To compare the efficiency of our method we compiled the expression $81*i+9*j+k-102$ using two other methods as well. The first method, the Constant Multiplier method, uses Symplify to extract and compile the constant multipliers that are used in the example. The second method, the Embedded Multipliers method, uses the multipliers which are embedded in the latest Virtex platforms. In all methods, we used the Symplify® 7.2 tool for synthesis and the Xilinx ISE 6.2 tool for hardware mapping. The results for the xc2v40 platform are presented in Table 6.1.

| *Method* | *Slices* | *Frequency* | *MUL18x18* |
|---|---|---|---|
| PSSA compilation | 37 | 250 Mhz | 0 |
| Constant Multipliers | 14 | 95 Mhz | 0 |
| Embedded Multipliers | 15 | 76 Mhz | 2 |

Table 6.1: Results for the example 6.11

We observe that the PSSA method is the faster. However, the price to be payed is in terms of resources that are used; more than twice of as for the other two methods. However, the total amount of slices used for our implementation is relatively small when we consider the total number of slices which we can find in a typical FPGA (e.g., 256 for xc2v40 to 33792 for xc2v6000). The Embedded Multipliers method is the slowest due to the placement of the multipliers in the Virtex platform. In this case, the routing of the signals plays a major

Listing 6.15: VHDL code of the example listed in 6.11

```
process (clk, rst)
  variable scan_v : INTEGER range 0 to 1023;
  variable k3_v : INTEGER range 0 to 1023;
  variable k3_i_v : INTEGER range 2 to 1023;
  variable scan_j_v : INTEGER range 9 to 1023;
  variable scan_k_v : INTEGER range 3 to 1023;
  variable scan_i_v : INTEGER range 162 to 1023;
 begin
  if rst='1' then
    scan_i_v := 162;
    scan_i <= scan_i_v;
    k3_i_v := 2;
    k3_i <= k3_i_v;
    scan_j_v := 9;
    scan_j <= scan_j_v;
    scan_k_v := 3;
    scan_k <= scan_k_v;
  elsif rising_edge(clk) then

    if end_loop_j then
      if end_loop_i then
        scan_i_v := 162;
      else
        scan_i_v := scan_i + 81;
      end if;
      scan_i <= scan_i_v;
    end if;

    if end_loop_j then
      if end_loop_i then
        k3_i_v := 2;
      else
        k3_i_v := k3_i + 1;
      end if;
      k3_i <= k3_i_v;
    end if;

    if end_loop_k then
      if end_loop_j then
        scan_j_v := 9;
      else
        scan_j_v := scan_j + 9;
      end if;
      scan_j <= scan_j_v;
    end if;

    if end_loop_k then
      k3_v := k3_i_v + 1;
    end if;
    if end_loop_k then
      scan_k_v := k3_v;
    else
      scan_k_v := scan_k + 1;
    end if;
    scan_k <= scan_k_v;
  end if;
end process;
scan <= scan_i+scan_j+scan_k -102;
```

role. The Constant Multipliers method is a pure combinational implementation. It does not use specific resources in the compilation process and it can be compiled at a greater speed comparing to the embedded multipliers approach.

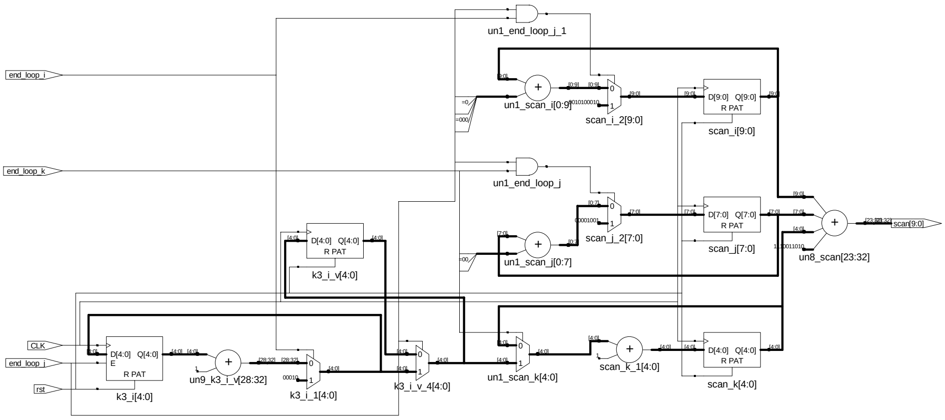


Figure 6.7: The RTL view of the PSSA from Figure 6.11

It is also interesting to see what the Expression Compiler does with the polytope given in Figure 6.2. Thus, the expression compiler compiles the linear expressions that make up the polytope and maps them onto a Virtex-II platform using Symplify. However, a large part of these expressions are simplified because they either are conditions over the program parameters or are for-loop upper and lower conditions. Hence, the remaining equations are:

$$Control\ Program = \begin{cases} 2*i-1 & \geq 0 \\ -i+2*DIV(i+1,2)-1 & \geq 0 \\ 2*DIV(M+1,2)-M-1 & \geq 0 \end{cases}$$

The DIV terms are simplified as shown in Figure 6.5 and the term $2*i-1$ is implemented as a shift plus a decrement operator. We found that we need only 5 slices to implement all the expressions, which are evaluated at approximately 200 Mhz. As a consequence, we should be able to integrate IP cores that run at 200 Mhz on a Virtex-II. However, signal routing in an FPGA negatively affects this number. In practice, we have found that embedding an IP core in a network synthesized by the LAURA tool has no problem running on a Virtex-II at 100 Mhz.

## 6.7 Conclusions

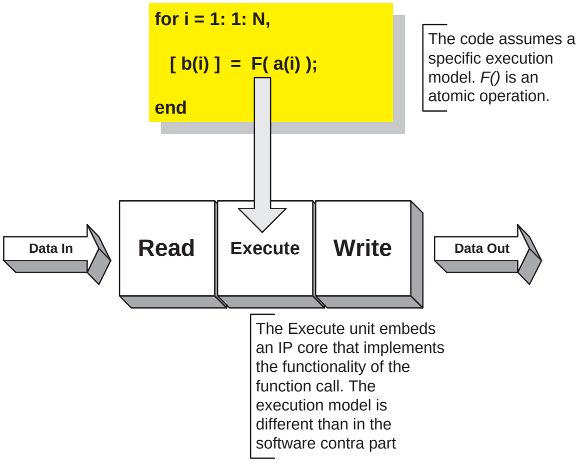
We have shown that expressions that are (pseudo-) affine can be converted efficiently to hardware using the Expression Compiler presented in this chapter. The Expression Compiler is needed in the LAURA methodology to make sure that the evaluation of polytopes in the Read and Write units happens faster than the evaluation of an IP Core embedded in the Execute unit. Only then, the dataflow in a CDFPN network is not obstructed by control needed to distribute the original application. The Expression Compiler first performs high-level optimizations based on DIV and MOD strength reduction operations and the Method of Differences technique. This step is followed by platform dependent optimizations using the Predicated

Static Single Assignment (PSSA) code. The PSSA form uses only additions, LUTs and conditional statements, resulting in an area/speed efficient hardware. Furthermore, the research community has shown that the PSSA form is well fitted to be mapped in reconfigurable hardware [74]. Expression Compiler is a part of the LAURA tool, helping in improving the quality of the synthesized network of processors. However, low-level optimization are not yet full implemented in our tool. But given the synthesis techniques as described in [86] the PSSA code can be further optimized for FPGA platforms.

Chapter 7

# IP Core Integration

In Figure 7.1, we show a small program, where $F()$ is a function call (a program itself), the details of which are not relevant here. The reason is that in the implementation of the main program a certain IP core will late care of the specific fine grain implementation of $F()$. In our model of execution, the Execute unit embeds the IP core that implements the functionality of a particular function $F()$ in a process of a process network that is derived from a given program.



```
for i = 1: 1: N,
  [ b(i) ]  =  F( a(i) );
end
```

The code assumes a specific execution model. *F()* is an atomic operation.

The Execute unit embeds an IP core that implements the functionality of the function call. The execution model is different than in the software contra part

Figure 7.1: Model of a computation vs. model of execution

This assumption, however, may lead to a mismatch between the non-functional behavior, e.g., timing, of $F()$ in the original program and the actual implementation . $F()$ in the original program is assumed to be a *function* (an atomic operation), while it may be non-atomic in the implementation. As a consequence, while the throughput in the underlying program depends on the execution time (latency) of $F()$, it may be much higher in the implementation when the IP core implementing $F()$ is (deeply) pipelined. To deal with this mismatch, we provide a feedback from the model of execution. This feedback is given by a *Profiler*, that relates non-atomic to atomic behavior. This chapter is broken down into two sections. In Section 7.1, we focus on the integration of an IP core in an Execute unit. In Section 7.2, we deal with the Profiler.

## 7.1 Embedding an IP Core

In the initial program a function call is an atomic operation, which may be assumed to have an instantaneous input-output relation or a certain non-zero execution time (latency). On the other hand, the corresponding function call in the actual implementation (the IP core) is a cascade of a number of atomic operations each having a certain latency. The maximum of these latencies defines the function's throughput, and their sum defines the function's latency. The IP cores that we consider are simple pipelined structures, which can be modelled as shift registers [87]. The Execute unit of a Virtual Processor represents the interface between an arbitrary IP core, and the Abstract Architecture.

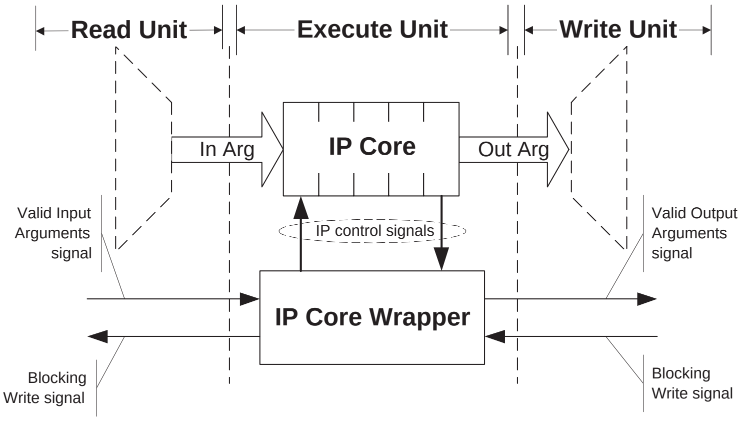


Figure 7.2: The Execute unit template

Figure 7.2 graphically shows the Execute unit template, and consists of two parts: an IP core and an IP core wrapper. The IP core implements a specific function call of the initial sequential program. The IP core wrapper interfaces the execution model of the IP core with the execution model of the Virtual Processor. The execution model of a Virtual Processor assumes a pipelined model of execution. The Virtual Processor pipeline can *stall* when an input FIFO buffer of the VP is empty (i.e., blocking read synchronization operation). Usually, the cause of this kind of stall is a *data hazard*. This type of stall is referred to as a *data hazard stall* [87]. In Section 7.2, we discuss how to detect and eliminate the data hazard stalls. The stalls due to a blocking read situation are detected by the Read unit, and they are communicated to the sequent units via *Valid Input Arguments signal*. The pipeline flow of a VP can be also held (stalled) when a blocking write situation occurs. In this situation an output FIFO buffer is full. The stalls due to a blocking write situation are detected by the Write unit, and they are communicated to the ahead units via *Blocking Write signal*.

The Execute unit differentiates between these stall situations, and it controls the IP core execution such that the tokens that are processed at the stall moment by the VP's pipeline flow are not damaged or discarded. Also, the Execute unit has to flow the pipelined execution model of the VP, synchronizing its execution with the Read unit and Write unit execution. The controlling of the IP core, and the VP internal intra-unit synchronization are handled by the IP core wrapper. Hence, the Execute unit integrates the IP core execution behavior into the general Virtual Processor pipelined execution model.

The valid input arguments signal generated by the Read unit indicates that the current input data ports (In Args) of the IP core hold valid data. This signal is propagated by the IP core wrapper to its Valid Output Arguments signal output with a fix latency. The latency is the latency of the embedded IP core. The Valid Output Arguments signal indicates to the Write unit that the output data ports (Out Args) of the IP core hold valid data that has to be written to a channel. The embedded IP core can have a run-time variation of its latency, e.g., ISA cores. Thus, the asserting of the Valid Output Arguments signal is handled by the IP core itself via the *IP control signals* shared between the IP core and IP core wrapper.

A blocking write situation implies a stalling of the entire VP pipeline flow. This situation is indicated by the Blocking Write signal. This signal is immediately forward by the IP core wrapper to Read unit, and to embedded IP core via IP control signals. How the Read unit and the embedded IP core react to this signal is discussed in the next section. At this moment, we can embed the IP cores that have the following execution models:

**Simple Pipelined IP cores** These are simple data streaming IP cores, that has the same execution behavior as a shift register.

**Variable latency IP cores** These cores are more complicated, usually Instruction Set Architectures, and require multiple clock cycles to execute a token. These cores has to provide an interface to communicate with the IP core wrapper via IP control signals.

### 7.1.1 Handling of Pipeline's Stalls

The stall due to a blocking read situation is eliminated by bubbling the VP's pipeline. In the bubbling technique, the control logic of the Read unit detects a hazard, and inserts no-operation (NOP) instructions into the pipeline. The NOP instructions travels through the VP's pipeline, flushing it. Hence we avoid artificial network deadlocks to occur in an Abstract Architecture. These network deadlocks can occur due to the data dependencies between the token currently processed and the token that the VP wants to read. However, the stall due to a blocking write situation cannot be eliminated by simply bubbling the VP's pipeline because we need to *store* the tokens that are currently in the VP's pipe until the VP's output ports are free again. The simplest solution is to *freeze* the VP's pipeline until the blocking write situation disappears. The freezing of a pipeline flow is done by simply gating its clock signal with an enable signal. Hence, when enable is asserted, the pipe is working, and when enable is not asserted the pipe holds the current state. A second solution is to provide enough storage capacity to store the tokens that are currently processed by the VP. This storage capacity can be a FIFO buffer placed between the outputs of the Execute unit and the inputs of the Write unit. The token is written into the FIFO buffer by the IP core wrapper, and read form FIFO buffer whenever there are no blocking write situations.

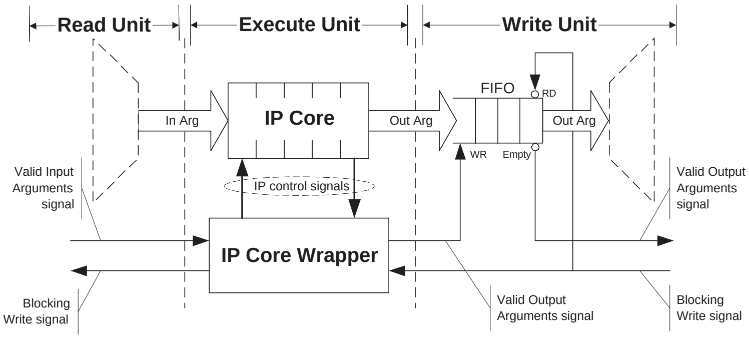


Figure 7.3: The solution to eliminate the blocking write stall

This solution is shown in Figure 7.3. The Read unit can either freeze or insert bubbles in the VP pipeline flow. The IP core wrapper generates and propagates the valid output arguments signal. The capacity of the FIFO buffer is selected to hold all the tokens that are processed by the IP core at any time instance, e.g., the FIFO buffer capacity is the equal with the number of stages of the embedded pipelined IP core.

## 7.2 Profiler

A CDFPN is an untimed model of computation. Its FPGA implementation is based on a timed model of execution. An atomic function $F()$ in the CDFPN may be mapped on a pipelined IP core in the implementation platform. As a consequence, the CDFPN predicted throughput may be far off the actual implementation throughput. The question, then, is whetter throughput prediction can be made more accurate. In this section we introduce the profiler, which we developed to estimate the utilization of embedded IP cores. The profiler analyzes the level of *data parallelism* that is mapped by the COMPAAN methodology onto a CDFPN process. Then, it measure the utilization of IP cores in the presence of the data parallelism, i.e., what is the ratio between NOP operations and token execution operations. The profiler hints a user of our methodology to perform certain network transformations as part of the design space exploration procedure for a particular problem. The amount of data parallelism in an algorithm is determined through data analysis. In our case, the amount of data parallelism mapped onto a process, and successively onto a Virtual Processor, can be determined by analyzing the FIFO capacities of the VP's self-loops as the self-loops being an effect of the data analysis performed by the COMPAAN tool.

The Virtual Processor has a pipelined execution model (Read, Execute, and Write stages). Many times the Execute unit embeds an pipelined IP core that is part of the VP pipeline flow. Given the Virtual Processor that embeds an IP core, we want to measure the utilization of the VP's pipeline flow, and possible to increase throughput of the given VP. A factor that affects the utilization of a VP's pipeline is the availability of data at an input of the processor. This is particulary in the case when self-loops are involved.

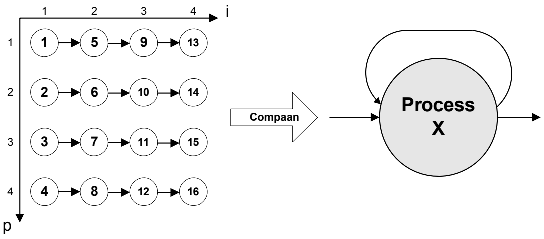


Figure 7.4: Self-loop example: arrows indicate data dependency

A particular function in *Process X* is fired 16 times. The 16 firings and their data dependencies are shown at the left of Figure 7.4. The data dependencies is detected by the COMPAAN compiler and shown as a self-loop of the *Process X*, see right part of Figure 7.4. There are two possible orderings. The first one is the firing order (1, 2, 3, 4, . . ., 15, 16); the second one is the order (1, 5,9, 13, . . ., 4, 8, 12, 16). For the first of these orders, the 4 firings in each $i$-column are independent, and the self-loop capacity is 4, because the 4 independent firings have to store their outputs until the next 4 independent firings are ready to read those results. For the second order, the 4 firings in each $p$-row are all dependency on each other, and the self-lop capacity is only 1.

Each of the firing orders has its own advantage and disadvantage, depending on the type of IP core embedded in the Virtual Processor that realizes the *Process X*. If the IP core is not pipelined, then the best firing order is the second one as the FPGA synthesis of the process uses a minimal capacity FIFO buffer implementation for the self-loop. However, when the embedded IP core is pipelined then the firing order affects in different ways the IP pipeline utilization.

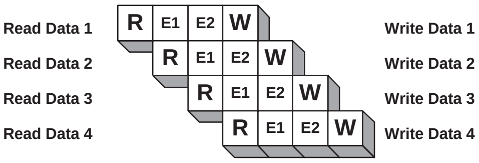


Figure 7.5: The optimal execution of a four stage processor pipeline

Now assume that the IP core that realizes the CDFPN *Process X* in Figure 7.4 has two pipeline stages. Thus, the total amount of pipeline stages of the Virtual Processor is rise up to four. For the first order, the pipeline achieves the maximum throughput because all the pipeline stages are filled with parallel data, as shown in Figure 7.5. The processor takes data either from a source (i.e., the first four iterations) or from itself (i.e. the remaining iterations). In the second example schedule, the pipeline is not fully used because the data generated by the processor is not yet available at its input due to the pipelining. The IP core is used for only 25% of its capacity, see Figure 7.6.

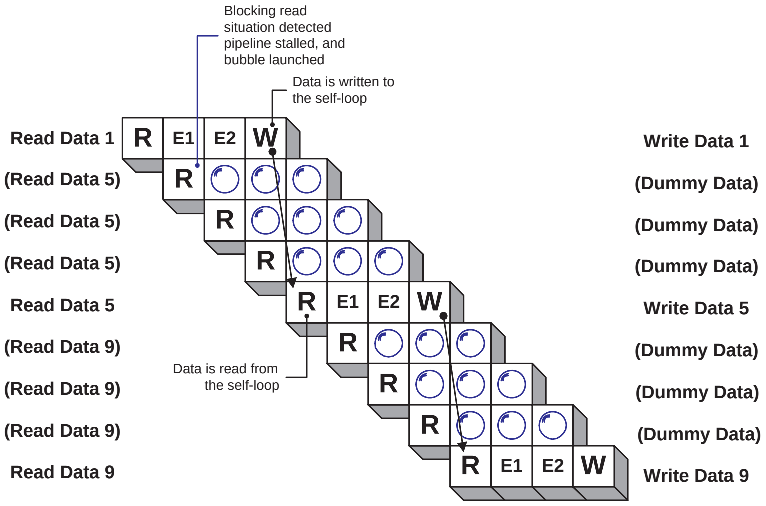


Figure 7.6: The broken execution of a four stage processor pipeline

As we can see, the presence of a self-loop influences the pipeline utilization of an Virtual Processor. The self-loop is the effect of a data dependency between various iteration points of the process (processor) iteration domain [7]. The order in which an iteration domain is scanned and the data dependencies can expose the inherent data parallelism. The amount of data parallelism of a variable is measured as the capacity of the associated self-loop FIFO buffer. When the scanning of the processor iteration domain coincide with the direction of the data dependency of a variable, then the capacity of the associated self-loop FIFO buffer is minimal and equal to one. The maximum capacity of a self-loop FIFO buffer is obtained when the scanning direction is orthogonal with the data dependency direction. Thus, the *minimum capacity* out of all self-loops FIFO buffer capacities of a processor is a key metric to measure the degree of data parallelism. In Chapter 5, we showed a procedure to determine the size of a self-loop at compile time.

Knowing the number of independent data and the embedded IP cores pipeline latency, we can hint the user of our methodology, who may take new design decisions skewing, loop swapping, and unfolding [70, 88]. Using the procedure to compute the size of the self-loop, and the given design decisions, we can close our tool chain such that we can perform a design space exploration of a given algorithm. In Figure 7.7, we show our tool chain flow with the profile feedback. We take an application written in Matlab and go through the COMPAAN/LAURA tool chain. The LAURA tool hints about the utilization of each individual processor of the Abstract Architecture. In [89], the authors have already shown how we can modify the Matlab programs to express unrolling and skewing as source to source operators. These operations are captured in the MATTRANSFORM tool. From MATTRANSFORM a modified version of the Matlab program is obtained and processed again by the COMPAAN/LAURA tool chain.

Nevertheless, quantitative simulation is needed to assess the final usefulness of the MAT-

TRANSFORM operations as the information offered by the profiler indicates only the maximum theoretical utilization of a VP's pipeline. The profiler does not give any performance information about the entire Abstract Architecture. However, we can repeat this procedure (profiler hints + simulation) an arbitrary number of times thereby exploring the design space for a particular application.

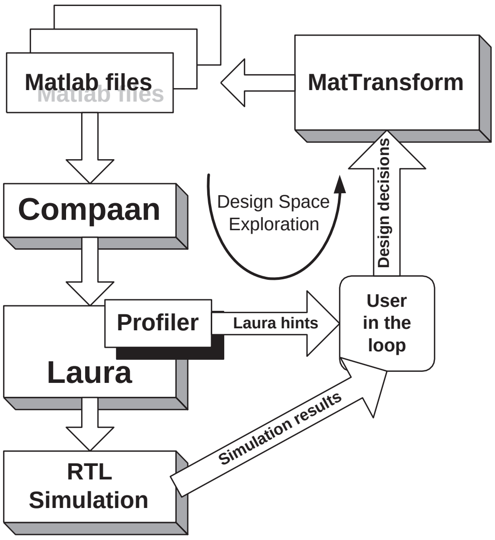


Figure 7.7: The modified COMPAAN/LAURA tool chain with feedback

### 7.2.1 Increasing the Pipeline Utilization of a Virtual Processor

In many cases, the existence of a self-loop buffer may limit the theoretical maximum pipeline utilization of the Virtual Processor. The self-loop is an effect of the data dependencies that exists between different iteration points of the iteration processor domain. This data dependency and the scanning order of the iteration processor domain expose the data parallelism that is mapped onto the processor. This amount of data parallelism available can be measured by checking the capacities of the self-loops buffers. The theoretical maximum pipeline utilization of a VP is derived from the amount of the parallel data that can feeded the VP's pipeline flow at a given time instance.

Consider a variable $\Delta$ that measure the data parallelism mapped onto a processor in the presence of self-loop buffers, and $FIFOsize_i$ the capacity of a VP self-loop buffer, where $i = 1..n$ the number of self-loop buffers of the VP in discussion. Then:

$$\Delta = \min(FIFOsize_i), i = 1..n \tag{7.1}$$

We can analyze now what are the effects of the amount of parallel data $\Delta$ over the VP's pipeline utilization. We found the following cases:

- When the amount of parallel data is larger than the number of VP's pipeline stages. In this case, we have a theoretical flooding of VP's pipeline flow with data. Thus, we can leave it as it is or we can increase the hardware resources to handle this data flooding. The unrolling operation of the MATTRANSFORM toolbox increases the hardware resources. A proposed unrolling factor is given by Equation 7.2, where $\Sigma$ is the number of VP's pipeline stages.

$$F_{unrolling} = \left\lceil \frac{\Delta}{\Sigma} \right\rceil \tag{7.2}$$

- When the amount of parallel data is smaller than the number of VP's pipeline stages. In this case, the VP's pipeline flow is hindered by possible frequent number of blocking read situations when the VP wants to access data from a self-loop buffer. Thus, the pipeline is stalled, and bubbles are launched, obtaining a lower theoretical maximum achievable utilization of the VP's pipeline. To increase the theoretical maximum achievable utilization of the VP's pipeline, we can apply different techniques to increase the amount of data parallelism. MATTRANSFORM provides the skewing operation for this purpose. We propose also to add multiple independent threads that are capable to be executed by the same Virtual Processor. A merging operation of the MATTRANSFORM can do this when the two merged processes function calls are pointing to the same IP core. An other approach is to add multiple independent threads as shown in Listing 7.2. The original code is shown in Listing 7.1. This solution is similar with the C-Slow techniques [90].

Listing 7.1: Example of a single threaded SANLP algorithm

```
for i = 1 : 1 : N,
    [x(i)] = F(x(i-1));
end
```

Listing 7.2: Source to Source transformation: adding independent threads of the algorithm shown in Listing 7.1

```
for i = 1 : 1 : N,
    for j = 1 : 1 : P,
        [x(i, j)] = IP_DCT(x(i-1, j));
    end
end
```

where $P = \Sigma - \Delta$, with $\Sigma$ and $\Delta$ previous defined.

- A special case is when $\Delta$ is one, and the embedded IP core latency is also one, resulting a value for $\Sigma$ to three. In this case, we have an instance of the classic case of *data hazard* [87].

  Consider a VP processor with a self-loop buffer. An instance of its execution is presented in Figure 7.8. In this instance, the processor reads tokens from the self-loop buffer, and writes tokens to the self-loop buffer. Thus, the token needed for the next IP core execution is available at its output ports, but not for the Read unit. The Read unit detects a blocking read situation, stalling the pipeline for two consecutive clock cycles, reading the token in the third. The bubbling of the pipe avoids the data hazard situation, however, it keeps the pipeline utilization low. The solution is to use *Data-Forwarding* [87] to solve this problem. The self-loop is replaced with a simple wire to route the data strait to the IP core data inputs from its outputs, as shown Figure 7.9. This technique is very beneficial, as the use of a wire increases the throughput of the processor requiring hardly any hardware resources.

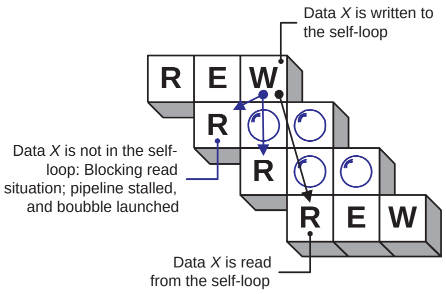

Data *X* is written to the self-loop
R
E
W
R
R
R
E
W
Data *X* is not in the self-loop: Blocking read situation; pipeline stalled, and boubble launched
Data *X* is read from the self-loop

Figure 7.8: Data Hazard in Virtual Processor template

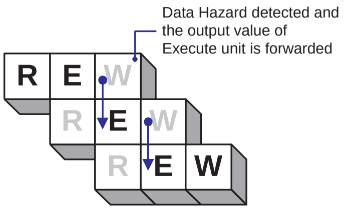

Data Hazard detected and the output value of Execute unit is forwarded
R
E
W
R
E
W
R
E
W

Figure 7.9: Data Forwarding in Virtual Processor template

### 7.2.2 Case Study

Consider the Matlab code that is processed by the COMPAAN compiler, as given in Listing 7.3. It shows two function calls (*bcell* and *icell*) surrounded by parameterized for-loops. The COMPAAN compiler generates a CDFPN based on the for-loops and the variables passed on to the function calls. What happens in the *bcell* and *icell* is irrelevant to the COMPAAN compiler, but not for the LAURA methodology. The network we obtain is shown in Figure 7.10. Each function call has become a separate process in the COMPAAN/LAURA methodology and both the *bcell* and *icell* processes have self-loops.

Listing 7.3: The algorithm using **bcell** and **icell** IP cores

```
for k = 1:1:T,
  for j = 1:1:N,
     [r(j,j), rr(j,j), a, b, d(k) ] =bcell( r(j,j), rr(j,j), x(k,j), d(k) );
     for i = j+1:1:N,
        [r(j,i), x(k,i)] =icell( r(j,i), x(k,i), a, b );
     end
  end
end
```

The *bcell* and *icell* Virtual Processors are embedding two deeply pipelined IP cores the **bcell** IP core (with 55 pipeline stages), and the **icell** IP core (with 44 pipeline stages), respectively. Let **N =7** and **T = 21**. For the case presented in Figure 7.10, our profiler reports for the smallest self-loop of **icell** a size five and for **bcell** a size 1. Given the deep pipelines of the **icell** and **bcell** IP cores, both of them are underutilized. The **bcell** theoretical maximum utilization is $1.75\%$ and for **icell** is $11.36\%$. Thus, we need to increase the amount of data parallelism for *bcell* and *icell* Virtual Processors by either using the skewing operation form MATTRANSFORM or adding multiple independent threads.

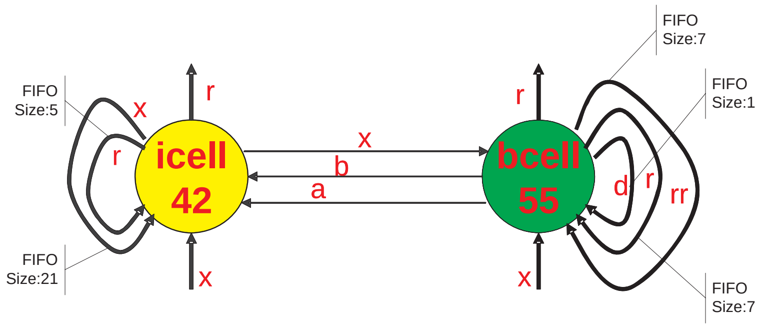


Figure 7.10: The CDFPN topology of the algorithm shown in Listing 7.3

First, we choose to explore the design decision of multiple independent threads, and then we evaluate the design decision of skewing. For each experiment, we derive a VHDL representation of the algorithm using the COMPAAN/LAURA tool chain. The VHDL is simulated to obtain quantitative data. The execution duration($No_{cycles}$) for each experiment is measured in clock cycles.

Each of the **icell**($No_{icell}$) and **bcell**($No_{bcell}$) operations contain $11$ and $16$ floating point operations, respectively. The average clock speed of our network mapped onto an FPGA platform is $100$Mhz (i.e., VIRTEX-II $6000$, Synplify Pro7.2, Xilinx ISE5.2). We use the next formula to compute how many million floating point operations per second (MFLOPS) can be achieved in each experiment.

$$\Lambda = \frac{No_{icell} * 11 + No_{bcell} * 16}{No_{cycles}} * 10^6 \tag{7.3}$$

### QR: Adding More QR Instances

For the original algorithm, the profiler reports that we have to add $57$ independent threads to the **bcell** IP core and $40$ independent threads to the **icell** IP core. Although the number of independents threads is large, we can find in practice algorithms that can have a such larger number of independent threads. In Section 7.2.3, I will give you such algorithm.

We choose to explore the space of independent instances by interleaving $1$, $10$, $20$, $30$, $40$ and $57$ independent threads to be executed using same hardware resources. In this case, a thread represents a complete instance of the algorithm. The results of this exploration are given in Figure 7.11. Adding multiple independent threads aim to increase the utilization of *bcell*.

We observe that the saturation point is marked by the running of $40$ independent threads, giving us a top performance of the implementation of $1683$ MFLOPS. At this point, the profiler reports $4.54$ times more parallel data than the **icell** can handle. Thus, adding more independent threads will not improve the performance of the system. Figure 7.11 shows the reaction of the system (in clock cycles) when multiple independent threads are used to boost the throughput of the algorithm. We can observe, a small difference between the situation when we run the algorithm alone or with $20$ independent threads. We basically replace the

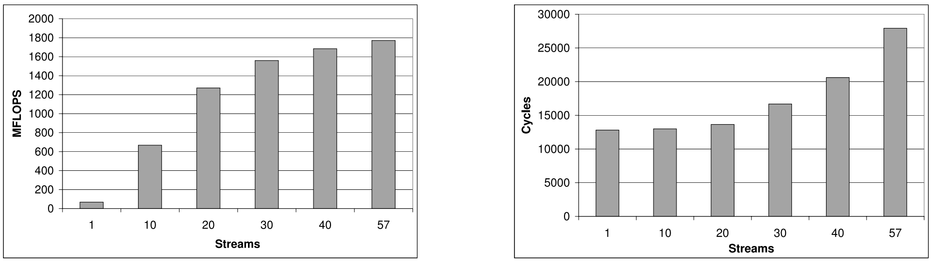


Figure 7.11: Experiments with multiple independent threads

stalling of the VP pipes with processing valid tokens. However, from 30 independent threads, we observe an increasing number of clock cycles needed to process the workload. The profiler indicates that after this point the **icell** is flooded with tokens.

We choose to unfold the **icell** twice, and four times respectively. This leads us to 1764 MFLOPS and 1767 MFLOPS, respectively. The first transformation gives us an additional 81 MFLOPS, the second one only an additional of 3 MFLOPS. It is obvious that the second operation is not as successful as the first one.

### QR: Skewing

A second approach to increase the data parallelism of an algorithm is by using skewing operation of MATTRANSFORM. The skew operation fills the VP's pipelines with parallel data that belongs to the same thread. The profiler indicates 7 independent data in **bcell** and 21 in **icell** after the skewing operation. Hence, we obtain a theoretical maximum pipeline utilization of 12.28% for **bcell** IP core, and 47.72% for **icell** IP core for a single threaded algorithm version. To achieve a higher throughput for our experiment, we must either increase the dimension of the input problem or add more independent threads. The total number of independent threads required to push up the throughput of the design is much smaller than in the case of non skewed algorithm. This is due to the already presence of the data parallelism in the skewed program. We choose to explore a number of 2, 4, 6, 7 and 10 independent threads mapped onto our architecture. The results are shown in Figure 7.12.

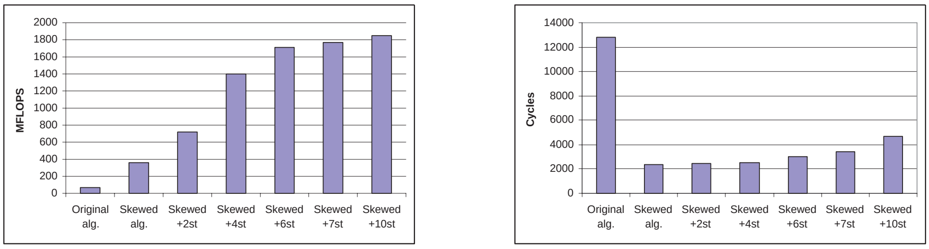


Figure 7.12: Experiments with skewing and multiple independent threads

From Figure 7.12, we can observe the improvement introduced by the skewing operation versus the original algorithm. Adding now more independent threads aims again to increase the utilization of the **bcell** IP core. However, in this case the number of additional independent threads added is much smaller than in the non skewed situation. Also in this case, adding threads flood the *icell* IP core with data. On the right part of Figure 7.12, we observe that adding the independent threads does not affect significatively the number of clock cycles needed by our architecture to complete the task. However, after adding $4$ independent threads the number of cycles starts to increase without a significative improvement in overall system throughput.

We duplicate the **icell** Virtual Processor hardware resources (using unrolling in MATTRANSFORM) of our implementation when the architecture handles $4$ independent threads. The profiler reports a $181\%$ utilization for the **icell** IP core just before unrolling. Performing the unrolling operation on the **icell** IP core twice gives us only $6$ MFLOPS more throughput compared with the non-unfolded algorithm. This shows us that the decision was not successfully. The reason for this is that the resulted two **icell** IP cores work in a mutually exclusive fashion, minimizing the effect of the unrolling operation.

### 7.2.3 Discussion

Based on the experimental results, skewing operation gives us from the beginning an important amount of parallel data. The large presence of parallel data allows us to obtain a higher computational throughput for our implementation without adding independent threads. It may be that the skewing operation doesn't achieve the maximum throughput of the cores and, therefore, solutions such as adding multiple independent threads may be required. However, we cannot determine at compile time if this is the case, and we cannot give a accurate hint to the user that this is the case. In practice, there are a number of algorithms that can accept adding multiple threads. One of these is JPEG 2000 image compression algorithm for large images. Instead to process the entire image, we can tile it and feed them to the JPEG 2000 hardware implementation. The tiles acts as multiple threads for the underling implementation. Usually, a tiling operation [64] is the basis of finding multiple threads for an application within the application.

The existence of large amount of data parallelism within the original algorithm helps the designer in achieving the maximum throughput with a minimum amount of independent threads added. Adding more independent threads to an architecture improves its throughput at the expense of additional memory requirements to realize the Abstract Architecture communication buffers. For example the self-loop of the **icell** IP core on $x$ variable needs $840$ locations when we add $40$ streams for the non skewed algorithm.

The data parallelism is also relevant for non pipelined cores such as ISA IP cores. In this case, we can unfold the respective IP core to take advantage of the parallel data. However, this has a higher implementation cost than using a pipelined IP core which is functional equivalent. The increasing of the cost comes from the implementation overhead introduced by the Virtual Processor wrapper, and additional communication channels introduced by the newly generated processors. The overhead introduced by the additional communication channel may kill the implementation performances as we discuss it in Chapter 8.

In practice, it is more likely that only a small number of independent instances of an algorithm need to be computed at the same time. Therefore, the skewed version is more appealing to start with in any design space exploration. The overloading of an IP core with many streams can be solved either by increasing the clock speed for that particular processor (i.e., using various clock domains) or by unrolling it. Applying high-level transformations, as hinted by the LAURA profiler, leads to an increased number of independent streams in our architecture. Nevertheless, quantitative simulation data is needed to assess their final usefulness.

## 7.3 Conclusion

In this chapter we presented a strategy to integrate particular types of IP cores in the Execute unit of Virtual Processors. The IP core are obtained from an IP library that is connected to our LAURA tool. The identification name of a function call in the algorithm helps us in recognition of the IP blocks from the IP library. The IP core wrapper bridges the gap between the function call execution model and the IP core execution model. However, we can handle only a small fraction of IP cores that exists. The best supported by our tool are the simple pipelined data streaming oriented IP cores.

A particular characteristic of the derived networks obtained from running the COMPAAN and LAURA tools, is the existence of self-loops. These self-loops have a large impact on the utilization of the IP cores and in the final implementation. This is especially the case when the IP cores are deeply pipelined. To improve the efficiency, the designer has to make design decisions like skewing, unrolling, loop swapping and data stripping. To help the designer in making these decisions, we have implemented the profiler in the LAURA tool. The profiler uses manipulation of polytopes to compute at compile time the size of self-loops, as it is described in Chapter 4. This size is indicative for the number of independent data available in an algorithm. The hints computed by the profiler help to steer design decisions. Doing this in an iterative manner, a designer can explore options to improve the throughput of the implementation. To improve the efficiency, the designer has to make transformations. These transformations can be expressed at the Matlab level using the MATTRANSFORM tool. Currently, the hints provided by the profiler needs to be manually expressed.

Chapter 8

# Case Studies

In this chapter we present the implementation of Matrix-Matrix multiplication, matrix QR factorization, and matrix Singular Value Decomposition (SVD) algorithms. These algorithms appear in an adaptive beam forming application [91]. We begin this chapter with a short introduction of beam forming (Section 8.1). The Matrix-Matrix multiplication is presented in Section 8.2. The matrix QR factorization is presented in Section 8.3, and the matrix SVD decomposition is presented in Section 8.4.

## 8.1 Subspace Tracking

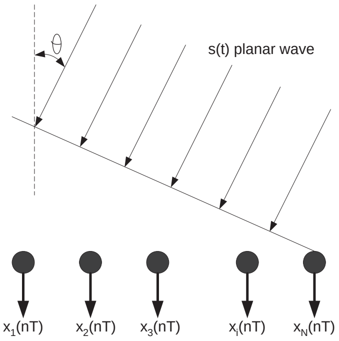


Figure 8.1: Planar wave example

Beam forming is technique that utilizes an array of sensor elements to receive a signal of interest that impinges on the array from a certain direction. Let the $1 \times N$ vector $x(nT)$ represent the set of output samples $x_i(nT),\ i = 1, 2, \ldots, N$ of a linear array of $N$ sensors at time $nT,\ n = \ldots, -1, 0, 1, \ldots$. Sensor $i$ receives a planar wave signal $s(t - \tau_i)$ from a far-end source signal $s(t)$. The direction of arrival od $s(t)$ is $\theta$, see Figure 8.1. The objective of the subspace tracking algorithm [92–94] is to detect and track the (slowly varying) angle $\theta$, and identify $s(t)$.

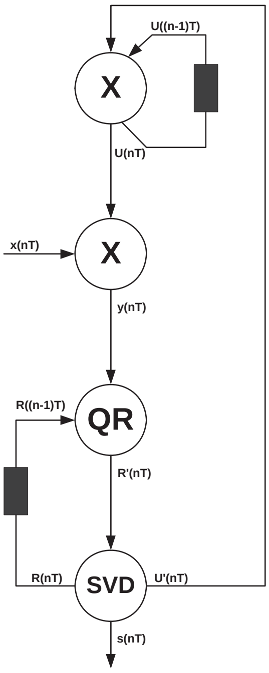


Figure 8.2: The Subspace Tracking Algorithm

A high-level flow-graph representation of the algorithm is show in Figure 8.2. In this flow-graph, $\mathrm{U}$ and $\mathrm{U}'$ are unitary matrices, $\mathrm{R}$ and $\mathrm{R}'$ are upper triangular matrices, and $x$ and $y$ are row vectors, all of appropriate dimensions. The black rectangles represent storage. The operators $\times$, QR, and SVD represent matrix-vector/matrix multiplication, matrix QR decomposition, and matrix Singular Value Decomposition (SVD), respectively. In the following sections, we give algorithms and implementation for the operators $\times$, QR, and SVD.

## 8.2 The Matrix-Matrix Multiplication Algorithm

Matrix multiplication is a core operation in many signal and image processing applications. Given two matrices $A = [a_{i,j}]$ and $B = [b_{j,k}]$ of dimension $N \times P$ and $P \times M$, respectively, the entries $c_{i,k}$ in the product matrix $C = A \times B$ are given as,

$$c_{i,k} = \sum_{j=1}^{P} a_{i,j} b_{j,k}$$

A possible algorithm is given in Listing 8.1. The out of the box implementation of the presented algorithm results in an Abstract Architecture network shown Figure 8.3. The Abstract Architecture has five communication channels of which one is an out-of-order with multiplicity communication type channel (i.e., ED_4), one is in-order with multiplicity (i.e. ED_3), and the rest of the channels are in-order communication types channels. The AA network also shows two processors that are the sources of the matrixes $A$ (i.e., ND_1) and $B$ (i.e., ND_2). The processor ND_4 wraps the IP core of the MAC function call, and processor ND_5 is the sink for the matrix $C$. Where IP core of the MAC function call implements $c = c + a * b$ operation. The processor ND_3 holds algorithm initialization values of $c$.

Listing 8.1: Matrix Multiplication algorithm

```
for i=1: 1: N,
    for j=1: 1: P,
        for k=1: 1: M,
            c(i,k) = MAC( c(i,k), a(i,j), b(j,k));
        end
    end
end
```

We want to implement the Matrix Multiplication algorithm using only FIFO channels as it should lead to a higher throughput realization. To achieve a network without reorder channels, we need to modify the data flow of the matrix-matrix multiplication algorithm. The reorder appears when the MAC core access multiple times the $b(j,k)$ entries. Hence, we have to store the variable $b(j,k)$ locally to avoid the reordering channel between processor ND_2 and ND_4. This local storage is converted by the COMPAAN compiler to an extra self-loop channel to the MAC core. The new architecture of the modified algorithm shown in Listing 8.2 is depicted in Figure 8.4.

Listing 8.2: Matrix Multiplication algorithm without reordering

```
for i=1: 1: M,
    for j=1: 1: N,
        for k=1: 1: P,
            [c(i,k), b(j,k)] = my_MAC( c(i,k), a(i,j), b(j,k));
        end
    end
end
```

The *MAC* core performs one addition and a multiplication of 32 bits operands, using the embedded Xilinx multipliers (e.g., for a 32 bit multiplication we use 3 hardware 18x18 multipliers). The *my_MAC* core has the same behavior as the function explained before, but, additionally, propagates the input $b(j,k)$ to its output to avoid the reorder channel.

In Table 8.1, we give results for three FPGA realizations for the Matrix Multiplication algorithm. The first two for the variant with the reordering memory, and the last one is for the version which is using only FIFOs. The difference between the first two variants is in the usage of the embedded multipliers to realize the read and write addresses for the reordering memory (i.e., the first version is using the embedded multipliers and the second one is using the method of differences (MoD)). The fastest implementation, in terms of cycles, is the third version. The fastest synthesis clock is the MoD version, however the number of clock cycles required for this implementation to complete a matrix multiplication is high.

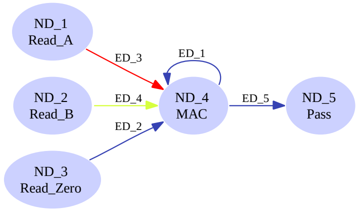


Figure 8.3: Matrix Multiplication: with one out-of-order communication type channel implementation version

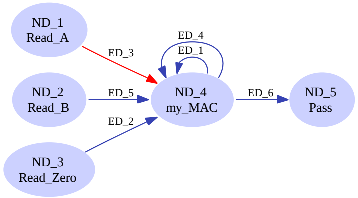


Figure 8.4: Matrix Multiplication: only FIFO implementation version

| | With 1 Reorder Channel and hardware multipliers | With 1 Reorder Channel using MoD | Only FIFOs |
|---|---|---|---|
| Minimum clock period | 10.9 ns (91 Mhz) | 10.1 ns (99 Mhz) | 13 ns (76 Mhz) |
| Number of RAM16x1D | 128 | 128 | 128 |
| Number of Block Rams | 1 of 56 | 1 of 56 | 2 of 56 |
| Number of Slices | 673 | 690 | 823 |
| Number of MULT18X18s | 5 | 3 | 3 |
| Cycles per Workload | 5514 | 5514 | 1011 |
| Throughput (MOPS) | 33.2 | 35.9 | 152.2 |

Table 8.1: FPGA mapping details for the Matrix Multiplication algorithm; $M = 10, N = 10, P = 10$

Thus, the FIFO implementation has the highest throughput (i.e., $152.2$ MOPS). The most efficient algorithm in terms of slices is the one which is using the embedded multipliers for the reordering channel implementation. At the opposite side lies the FIFO implementation which uses $823$ slices to implement the Matrix Multiplication algorithm.

### 8.2.1 Discussion Matrix-Matrix Multiplication Implementation

The Matrix Matrix multiplication algorithm is using a small IP core for real, fix point numbers. This core is not larger than three hardware multipliers and some registers associated to the accumulator part. In this case, the control overhead introduced by our methodology is large. The overhead is caused by the control needed for the realization of the channels. We can easily show that this is the case by checking the amount of resources needed for in-order version and for out-of-order versions. The addition of the additional FIFO adds more than $100$ extra slices and an extra memory block for the in-order version.

There is a trade off between the resources consumed for the realization of the communication and the size of the IP cores embedded in our networks. For example, we can embed a IP core that implements the MAC function call for complex floating point numbers. In this case we deal with a coarse grain IP core that is larger than the control overhead. I.e., for this IP core we need $4$ floating point multipliers and $4$ floating point adders. A floating-point single precision (i.e., $32$ bit) adder needs $183$ slices, and a floating-point single precision multipliers needs $221$ slices. Thus, for a complex floating point IP core that implements the MAC function call requires $1616$ slices compared with approximatively $800$ slices needed by the CDFPN control.

## 8.3 The QR Factorization Algorithm

QR decomposition is a matrix computation algorithm commonly used to solve an over-specified set of linear equations in a least squares sense. QR decomposes a matrix $X$ into a product of an orthogonal matrix $Q$ and an upper triangular matrix $R$ [95]. QR decomposition uses elementary Givens Rotations method [91] as basic operators. This algorithm is widely used in signal processing applications [52].

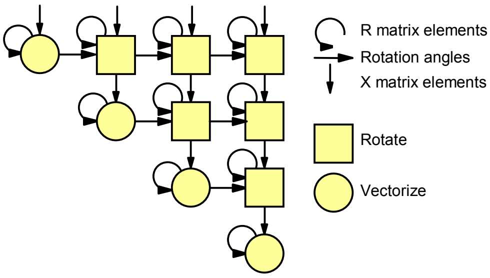


Figure 8.5: QR data dependencies

As shown in Figure 8.5, QR employs two operations: *Vectorize* and *Rotate*. Vectorize takes a vector $[x\ r]$ formed by an element of $X$ and an element of $R$ and rotates it over an angle $t$ to $[0\ r']$. The Rotate operation takes a similar vector $[x\ r]$ and rotates it over an angle $t$ previously calculated by a Vectorize operation. The Matlab code written as input to the COMPAAN compiler is shown in Listing 8.3, and describes the Givens rotations calculations. The initializations and terminations (sources and sinks) are not shown.

Listing 8.3: QR factorization algorithm

```
for k = 1: 1: K,
    for j = 1: 1: N,
        [r(j,j),t] = Vectorize( r(j,j),x(k,j));
        for i = j+1: 1: N,
            [r(j,i),x(k,i)] = Rotate(r(j,i),x(k,i),t);
        end
    end
end
```

In this algorithm, the iterator $j$ counts down the rows of the $R$ matrix, the iterator $i$ enumerates the entities in a row of $R$, and $k$ counts down the rows of the matrix $X$. The loop bounds $K$ and $N$ are constant parameters whose values are the number of QR updates, and the size of the square matrix $R$, respectively. The process network produced by the COMPAAN compiler for this code consists of five interconnected nodes (see shown Figure 8.6). The algorithm requires 11 communication channels in which one is of type in-order with multiplicity (IOM+) (i.e., ED_9) and the rest of the channels are of type in order (IOM-).

Computation nodes are nodes ND_4 and ND_3,the *Vectorize* node (ND_3) and the *Rotate* node (ND_4). The other nodes are data sources (ND_1 and ND_2) and terminators (ND_5).

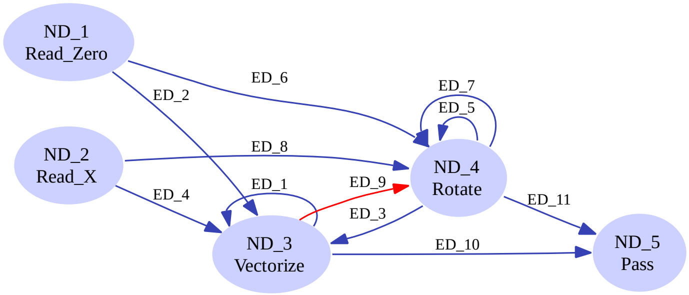


Figure 8.6: The QR Process Network

| | |
|---|---|
| Minimum clock period | 9.6 ns (103 Mhz) |
| Number of RAM16x1D | 1024 |
| Number of Slices - QR Control | 4713 |
| Number of Slices - IP cores | 3442 |
| **Number of Slices - Total** | **8155** |
| Cycles per Workload | 12808 |

Table 8.2: FPGA mapping details for the QR implementation; $N = 7, K = 21$

The implementation results of the QR algorithm are shown in Table 8.2. The RAMB16x1D components are used for the implementation of the FIFO channels. We use 4713 slices for the CDFPN control. The board used is an Annapolis WildCard-II which embeds an xc2v3000fg676-6 Xilinx device. Additionally, the IP cores of *Vectorize* and *Rotate* take another 3442 slices. Thus the total number of slices used for a complete QR implementation is 8155 slices. The *Vectorize* and *Rotate* processors are pipelined and have 55 and 42 stages, respectively.

### 8.3.1 Discussion Matrix QR Factorization Implementation

The communication network for the QR algorithm is more complex than the one for the Matrix Matrix multiplication algorithm. Form Table 8.2, we can observe that the sizes of the IP cores are in the same range (in terms of resources) as for the control. The result is a well balanced architecture that allows us to achieve a good synthesis frequency. High level algorithmic transformations may be applied to reduce the total number of cycles per workload. For example, the skewing transformation reduces the total number of clock cycles to $2405$ without introducing extra channels or processors.

## 8.4 The Matrix SVD Decomposition Algorithm

The singular value decomposition (SVD) [95] of a real square matrix $A \in \mathbb{R}^{n \times n}$ is a product decomposition, consisting of two orthogonal matrices U and V, and a non-negative diagonal matrix $\Sigma = diag(\sigma_1, \ldots, \sigma_n)$ such that:

$$A = U \Sigma V^T \tag{8.1}$$

The $\sigma_i$ are the singular values of $A$.

There are several ways to compute the SVD. For parallel implementation, the Jacobi method [96] is better suited for this task. Luk's algorithm [96] is based on the classical Jacobi algorithm and computes the SVD of a symmetric matrix $A$ by a series of Jacobi rotations. A Jacobi rotation, denoted $J(p, q, \theta)$, is an orthogonal matrix which equals the identity matrix except for the four entries:

$$\begin{array}{ll} J_{pp} = \cos(\theta) & J_{pq} = \sin(\theta) \\ J_{qp} = -\sin(\theta) & J_{qq} = \cos(\theta) \end{array} \tag{8.2}$$

The Jacobi algorithm consists of a sequence of operations of the form:

$$A_{k+1} = J(p, q, \theta)^T A_k J(p, q, \theta) \tag{8.3}$$

where the angle $\theta$ is chosen such that the elements $a_{pq}$ and $a_{qp}$ of $A_k$ are annihilated. For this to be accomplished $\theta$ must satisfy:

$$\tan(2\theta) = \frac{a_{pq} + a_{qp}}{a_{qq} - a_{pp}} \tag{8.4}$$

For non symmetric matrices, the algorithm is slightly different and is know as Kogbetliantz' algorithm [96]. In this algorithm the angle in the left-hand side rotation may be different from the angle in the right-hand side rotation. Hence, 8.3 becomes 8.5.

$$A_{k+1} = J(p, q, \theta_1)^T A_k J(p, q, \theta_2) \tag{8.5}$$

and the angles $\theta_1$ and $\theta_2$ satisfy:

$$\begin{array}{rcl} \tan(\theta_1 + \theta_2) & = & \dfrac{a_{pq} + a_{qp}}{a_{qq} - a_{pp}} \\ \tan(-\theta_1 + \theta_2) & = & \dfrac{a_{pq} - a_{qp}}{a_{qq} + a_{pp}} \end{array} \tag{8.6}$$

The Matlab program that computes the SVD of a matrix $M \times M$ matrix $A$ is listed in Listing 8.4 according to the the odd-even Kogbetliantz algorithm [96]. Function *Angle* computes the angles $\theta_1$ and $\theta_2$ for the planar operators $R(\theta_1)$ and $R^T(\theta_2)$, where $R()$ is a planar Givens Rotation. The function *RotRow* rotates the vector $(a_{i,j} \; a_{i+1,j})$ to $(a'_{i,j} \; a'_{i+1,j}) = (a_{i,j} \; a_{i+1,j}) R(\theta_1)$, and the function *RotColumn* rotates the vector $\binom{a_{i,j}}{a_{i,j+1}}$ to $\binom{a\text{‘}_{i,j}}{a\text{‘}_{i,j+1}} = R^T(\theta_2)\binom{a_{i,j}}{a_{i,j+1}}$.

Figure 8.7 shows the corresponding the PN topology. The network encompasses $54$ IOM-, $30$ OOM-, and $4$ IOM+ communication channels. We shown in Section 8.2 that the out-of-order communication type channels have large latency, stepping down the throughput performance of an architecture. The out-of-order communication type channel appears due to ordering in which data is produced and consumed. Hence, the dataflow through the architecture is hindered by the reorder mechanisms. The ideal case is to remove these reorder

Listing 8.4: The ODD-EVEN SVD

```
for stage = 1: 1: N,
    for i = 1: 2: M-1,
        [th1(i), th2(i)] = Angle(a(i,i), a(i,i+1), a(i+1,i), a(i+1,i+1));
    end

    for i = 1: 2: M-1,
        for j = 1: 1: M,
            [a(i,j), a(i+1,j)] = RotRow(th1(i), a(i,j), a(i+1,j));
        end
    end

    for i = 1: 2: M-1,
        for j = 1: 1: M,
            [a(j,i), a(j,i+1)] = RotColumn(a(j,i), a(j,i+1), th2(i));
        end
    end

    for i = 2: 2: M-2,
        [th1(i), th2(i)] = Angle(a(i,i), a(i,i+1), a(i+1,i), a(i+1,i+1));
    end

    for i = 2: 2: M-2,
        for j = 1: 1: M,
            [a(i,j), a(i+1,j)] = RotRow(th1(i), a(i,j), a(i+1,j));
        end
    end

    for i = 2: 2: M-2,
        for j = 1: 1: M,
            [a(j,i), a(j,i+1)] = RotColumn(a(j,i), a(j,i+1), th2(i));
        end
    end
end
```

mechanisms and obtaining a streaming architecture that uses only in-order communication type channels.

The algorithm presented in Listing 8.4 results in an implementation that requires many reorder mechanisms that regulates the dataflow. These reorder mechanisms are mainly between the *RotRow* functions and *RotColumn* functions. Observe that the *RotRow* functions iterates over the elements of the matrix $A$ elements in a row wise order, while the *RotColumn* functions iterates over the elements of the matrix $A$ in a column wise order. The *RotColumn* functions are using the tokens produced by the *RotRow* functions. Thus, reorder mechanisms are required to hold the tokens produced by the *RotRow* functions (an entire iteration over a row of matrix $A$) until the *RotColumn* functions consumes them. A tiling source-to-source transformation [64] changes the order in which the *RotColumn* iterates over the matrix $A$ elements. Listing 8.5 shows the result of the tiling source-to-source transformation applied on the odd-even SVD algorithm shown in Listing 8.4. This new version of the odd-even SVD algorithm forces the *RotColumn* functions to closely flow the order in which the *RotRow* functions are iterating over the elements of $A$ matrix. Hence, removing the need of reorder mechanisms between *RotRow* functions and *RotColumn* functions. The tiling operation is applied to both RotColumn functions by introducing the additional nested for-loop with the index $tj$. The tiled odd-even SVD algorithm implementation encompasses $95$ communication channels of which $89$ are IOM-, $2$ OOM-, $2$ IOM+, and $2$ OOM+.

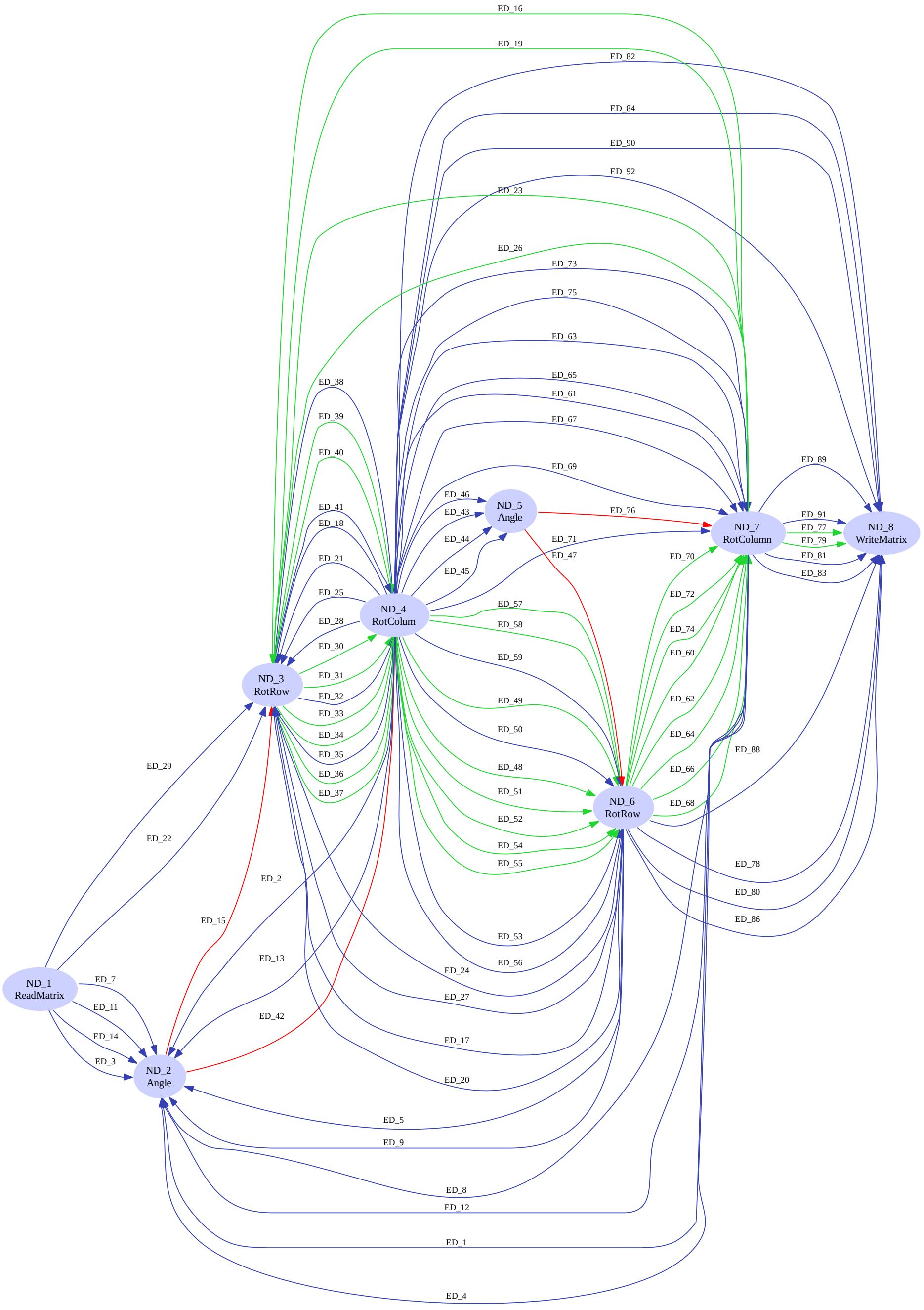


Figure 8.7: The Odd-Even SVD Process Network

Listing 8.5: The TILED ODD-EVEN SVD

```
for stage = 1: 1: N,
    for i = 1: 2: M-1,
        [th1(i), th2(i)] = Angle(a(i,i), a(i,i+1), a(i+1,i), a(i+1, i+1));
    end

    for i = 1: 2: M-1,
        for j = 1: 1: M,
            [a(i,j), a(i+1,j)] = RotRow(th1(i), a(i,j), a(i+1,j));
        end
    end

    for tj = 1: 2: M-1,
        for i = 1: 2: M-1,
            for j = tj: 1: min(M,tj+1),
                [a(j,i), a(j,i+1)] = RotColumn(a(j,i), a(j,i+1), th2(i));
            end
        end
    end


    for i = 2: 2: M-2,
        [th1(i), th2(i)] = Angle(a(i,i), a(i,i+1), a(i+1,i), a(i+1, i+1));
    end

    for i = 2: 2: M-2,
        for j = 1: 1: M,
            [a(i,j), a(i+1,j)] = RotRow(th1(i), a(i,j), a(i+1,j));
        end
    end

    for tj = 1: 2: M-1,
        for i = 2: 2: M-2,
            for j = tj: 1: min(M,tj+1),
                [a(j,i), a(j,i+1)] = RotColumn(a(j,i), a(j,i+1), th2(i));
            end
        end
    end
end
```

### 8.4.1 Discussion Matrix SVD Decomposition Implementation

We run both odd-even SVD algorithm variant implementations for a $10 \times 10$ matrix. The implementation results are presented in Table 8.3. The number of cycles needed for one $10 \times 10$ SVD update is $405$ cycles in the case of the original odd-even implementation. The tiled implementation needs $279$ cycles to compute an SVD update. Hence, the tiled algorithm is $30\%$ faster than the standard version. A significative improvement is obtained also in the memory requirements for each of the implementations (i.e., $12160$ bytes for odd-even SVD implementation versus $7560$ bytes for the tiled version). The memory requirement for an implementation is the summation of the all channels capacities.

Although the tiled odd-even SVD implementation has more communication channels, the total number of hardware resources used (slices and RAMB16 memories) is less than in the case of the odd-even SVD implementation. The low figures for the synthesized clock frequencies is a result of the heavy communication topology of the CDFPN networks. As a rule of thumb, the architecture performance decreases in strong relation with the number of channels of the input PN. This affirmation is also sustained if we check the QR realization (Table 8.2) versus the SVD realization. The QR implementation uses almost as many

| | Odd-Even SVD implementation | Tiled Odd-Even SVD implementation |
|---|---|---|
| Minimum clock period | 17 ns (58 Mhz) | 21 (49 Mhz) |
| Number of Block RAMs | 74 | 72 |
| Number of Slices | 9040 | 4848 |
| Clock Cycles | 405 | 279 |
| Memory Requirements(bytes) | 12160 | 7560 |
| Channels (IOM-/OOM-/OOM+/IOM+) | 88 (54/30/-/2) | 95 (89/2/2/2) |

Table 8.3: Results for SVD algorithm

slices as in the case of the original SVD implementation. However, we obtain a much lower synthesized clock frequency in the case of SVD implementation. The difference is in the number of communication channels that QR architecture has (i.e., $11$) versus the number of communication channels the SVD architecture has (i.e., $95$).

## 8.5 Discussion

The Matrix-Matrix multiplication algorithm is edificatory for the usage of our methodology in implementing algorithms with low-grain IP cores, and a relatively uncomplicated data-flow. Such algorithms are Discrete Fast Fourier Transformation, Finite Impulse Filters, Convolutions cores, etc. These algorithms are more optimal implemented using other synthesis techniques [21, 30, 31, 34]. The synthesis methodology used by us results in a large amount of control that is required to handle the distributed nature of our architecture.

An optimal result is achieved for algorithms that embeds IP cores that are more coarse grained. Such case is the matrix QR factorization algorithm. Also the communication between these IP cores is more complex, requiring various techniques to increase the IP cores computational efficiency (see Chapter 7). In particular cases, we can obtain efficient automatical generated implementations [97] that have performance figures close to hand crafted implementations [52]. We have again coarse grained IP cores in the case of the matrix SVD decomposition. The large number of communication channels affects the final performance figures of our implementation. However, there are techniques [7] to merge these channels, with a potential to simplify the implementation. It is desirable to avoid as much as possible large number of channels that are out-of-order. In general, such channel stalls the execution of the processors that are accessing them. Source-to-source transformations such as skewing, loop swap and tiling [64] optimize the data-flow of an algorithm, removing the need of re-order mechanisms. In the tiled odd-even SVD implementation, the tiling operation removed a large amount of out-of-order communication type channels, obtaining improved performance figures (throughput and resources consumption) relatively to the original implementation of odd-even SVD algorithm.

As we step up in the level of abstraction of the description of an algorithm, the IP cores becomes more and more complex. And, therefore, the data-flow is less complicated. Such example is M-JPEG application explored in [8]. However, when the final architecture performances figures are not an issue, our methodology delivers fast a parallel implementation mapped onto an FPGA platform.

# Chapter 9

# Conclusions

In this dissertation, we presented a compilation technique that takes applications specified as COMPAAN Data Flow Process Networks and produces optimized FPGA implementations. The optimizations performed include usage of IP cores, synthesis of the communication control, and finding a capacity for the communication memory.

Chapter 2 introduces LAURA's Virtual Processor and the first step of our methodology called PN to Abstract Architecture. The PN to Abstract Architecture step constructs an Abstract Architecture out of a COMPAAN Data Flow Process Network MoC using a one-to-one mapping. The Abstract Architecture is a set of hierarchical interconnect modules representing an architecture. Each module embeds an IP core that is wrapped by a module wrapper to isolate the computation from communications.

The construction of the Abstract Architecture consists of a semantic mapping and a topological mapping. The topological mapping generates the network, while the semantic mapping translates a CDFPN process to a Virtual Processor. The synthesis of each communication channel of the Abstract Architecture is described in Chapter 4. Upper bounds for FIFO capacities in the communication channels of the Abstract Architecture are derived using bounding box techniques presented in Chapter 5. Self-loop channel FIFO capacities are accurately estimated using Ehrhart polynomials [42]. The amount of memory estimated by using the bounding box techniques is dependent on the way of describing the CDFPN as an imperative program. Hence, source-to source transformations that increase data locality in a sequential application can diminish the memory requirements for a channel, i.e., loop tiling [64] and loop fusion [65].

In the PN mapping to Abstract Architecture, each process is mapped to a *Virtual Processor* consisting of a Read, an Execute, and a Write unit. The Read and Write units use a local controller to execute a particular control program that is derived by the COMPAAN tool. Because of the inherent complexity of this control, deriving an efficient controller implementation (both in terms of area and speed) is mandatory to obtain a good performance. We studied three different methodologies for an efficient derivation of the Read and Write unit control in Chapter 3.

The Abstract Architecture is synthesized to a network of synthesizable processors that is implemented in an FPGA execution platform. The FPGA implementation of a network of synthesizable processors is a Globally Asynchronous Locally Synchronous (GALS) system [39]. GALS systems eliminate the need for careful design and fine-tuning of a global clock distribution network. Besides energy conservation and global clock distribution, designers are now seriously exploring opportunities for reusing IP cores. To synthesize an Abstract Architecture to a network of synthesizable processors, a number of techniques are employed. Chapter 6 shows how expressions that are (pseudo-) affine can be converted efficiently to hardware using the Expression Compiler presented in this chapter. The Expression Compiler is needed to make sure that the evaluation of polytopes in the Read and Write units happens faster than the evaluation of an IP Core embedded in the Execute unit. Only then is the dataflow in a CDFPN network not obstructed by the distributed control. The Expression Compiler first performs high-level optimizations based on DIV and MOD strength reduction operations and the Method of Differences technique. This step is followed by platform dependent optimizations using the Predicated Static Single Assignment (PSSA) code . The PSSA form uses only additions, LUTs and conditional statements, resulting in an area/speed efficient hardware.

Our methodology synthesizes only the control of a CDFPN into an Abstract Architecture. The functionality of a processor is implemented by embedding an IP core. In Chapter 7, an IP core wrapper is introduced to act as an interface between the synchronous IP core and the asynchronous network. Hence, we do not generate an implementation for a particular software function call, but we use IP cores to get the functionality of the sequential function. We can predict a maximum theoretical throughput of an embedded IP core at compile-time by using the profiler presented in Chapter 7. The profiler gives hints that may increase the maximum theoretical throughput of the embedded IP cores. The hints are applied using the MATTRANSFORM tool box.

Finally, three test cases are used to benchmark the present methodology. We consider the design of parts of a complex application (Subspace Tracking). The parts we consider are three key kernels: Matrix-matrix multiplication, matrix SVD decomposition, and matrix QR factorization. For the Matrix-matrix multiplication kernel, we investigated the control overhead introduced by the presented methodology in the case of embedding simple IP cores. At the opposite pole lays the matrix QR factorization that uses complex computational IP cores. Here, we investigated the control overhead introduced by the present methodology when we embed non trivial IP cores. The matrix SVD decomposition kernel has a complex communication network. Here, we investigated the influence of such complex communication networks on the FPGA implementation. All these case studies and their conclusions are presented in Chapter 8.

The work presented in this dissertation enables the conversion of a CDFPN model into an FPGA implementation. The conversion of an imperative program into a CDFPN model is addressed by Rijpkema et al. [5, 6], and extended by Turjan [98] and Stefanov [8]. The conversion of the imperative program into a parallel description is prototyped in the COMPAAN tool. The conversion of the parallel description of an imperative program into an FPGA implementation is prototyped in the LAURA tool. Together, the LAURA and COMPAAN tools, provide a flow to *prototype* an FPGA implementation of an imperative program. The set of imperative programs is confined to the class of piece-wise affine nested loop programs. This restriction still allows to study the kernels of the applications that belong to the domains of digital signal processing and multimedia.

# Bibliography

[1] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[2] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[3] Bart A.C.J. Kienhuis. *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. PhD thesis, Delft University of Technology, The Netherlands, January 1999.

[4] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[5] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Nested Loop Alogorithms. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[6] Edwin Rijpkema. Modeling Task Level Parallelism in Piece-wise Regular Programs, 2002. PhD thesis, Leiden University, The Netherlands.

[7] Alex Turjan. Compiling Nested Loop Programs to Process Network, 2007. PhD thesis, Leiden University, The Netherlands.

[8] Todor Stefanov. Converting Weakly Dynamic Programs to Equivalent Process Network Specifications, 2004. PhD thesis, Leiden University, The Netherlands.

[9] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[10] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[11] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Multi-processor system design with espam. In *CODES+ISSS '06: Proceedings of the 4th international conference*

*on Hardware/software codesign and system synthesis*, pages 211–216, New York, NY, USA, 2006. ACM.

[12] G. Spivey, S. S. Bhattacharyya, and K. Nakajima. Logic foundry: Rapid prototyping for FPGA-based DSP systems. *EURASIP Journal on Applied Signal Processing*, 2003(6):565–579, May 2003.

[13] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'02)*, pages 231–236, Kyoto, Japan, October 2-4 2002.

[14] D. Lyonnard et al. Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip. In *Proc. 38th Design Automation Conference (DAC'2001)*, Las Vegas, USA, June 18-22 2001.

[15] P. Paulin, C. Pilkington, M. Langevin, E. Bemsoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G Beltrame, V. Gagne, and G Nicolescu. Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia. *IEEE Trans. on VLSI Systems*, 14(7), July 2006.

[16] A. Jerraya, A. Bouchhima, and F. Petrot. Programming Models and HW-SW Interfaces Abstraction for MultiProcessor SoC. In *Proc. 43th Design Automation Conference (DAC'06)*, San Francisco, USA, July 24-28 2006.

[17] M.J. Rutten et all. A Heterogeneous Multiprocessor Architecture for Flexible Media Processing. *IEEE Design & Test of Computers*, 19(4), 2002.

[18] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, pages 1363– 1375, November 2004.

[19] F. Balarin, E. Sentovich, M Chiodo, P. Giusto, H. Hsieh, B Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, 1997.

[20] Michael C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[21] M Haldar, A Nayak, A Choudhary, and P Banerjee. A System for Synthesizing Optimized FPGA Hardware from MATLAB. In *Proc. Int. Conf. on Computer Aided Design*, San Jose, CA, November 2001.

[22] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. From loop transformation to hardware generation. In *Proceedings of the 17th ProRISC Workshop*, pages 249–255, Veldhoven, 11 2006.

[23] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[24] Jurgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14:297–332, February 1993.

[25] Hritam Dutta, Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. Efficient control generation for mapping nested loop programs onto processor arrays. *J. Syst. Archit.*, 53(5-6):300–309, 2007.

[26] Sanjay V. Rajopadhye and Richard Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14(2):163–189, 1990.

[27] Patricia Le Moenner and et al. Generating regular arithmetic circuits with alphard.

[28] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[29] J. L. Van Meerbergen, P. E. R. Lippens, W. F. J. Verhaegh, and A. Van der Werf. Phideo: high-level synthesis for high throughput applications. *J. VLSI Signal Process. Syst.*, 9(1-2):89–104, 1995.

[30] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. PICO: Automatically Designing Custom Computers. *Computer*, 35(9):39–47, 2002.

[31] http://www.synfora.com.

[32] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[33] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. *fccm*, 00:49, 2000.

[34] Ian Page. Constructing hardware-software systems from a single description. In *Journal of VLSI Signal Processing, 12(1):87–107*, 1996.

[35] Andre Nieuwland, Jeffrey Kang, O. P. Gangwal, R. Sethuraman, N. Busa, K Goosens, R. P. Llopis, and P. Lippens. *C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems*. Kluwer Academic Publishers, 2002.

[36] M. Diaby, M. Tuna, J.-L. Desbarbieux, and F. Wajsburt. High level synthesis methodology from c to fpga used for a network protocol communication. In *RSP '04: Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*, pages 103–108, Washington, DC, USA, 2004. IEEE Computer Society.

[37] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, and W. J. M. Smits. Cosy communication ip's. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 406–409, New York, NY, USA, 2000. ACM.

[38] Prophid: a heterogeneous multi-processor architecture for multimedia. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 164, Washington, DC, USA, 1997. IEEE Computer Society.

[39] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Dept. of Computer Science, Stanford Univ., 1984.

[40] M Ben-Ari. *Principles of Concurrent Programming* . Prentice Hall, 1982.

[41] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. An Integer Linear Programming Approach to Classify Communication in Process Networks. In *8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Amsterdam, September 2–3 2004.

[42] Eugène Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*. Birkhäuser Verlag, Basel, international series of numerical mathematics vol. 35 edition, 1977.

[43] R. L. Walke, R. W. M. Smith, and G. Lightbody. 20GFLOPS QR processor on a Xilinx Virtex-e FPGA. In *proceedings of SPIE advanced signal*, 1999.

[44] Kurt Konolige. Small Vision Systems: Hardware and Implementation. In *Eighth International Symposium on Robotics Research*, Hayama, Japan, October 1997.

[45] E. Mémin and T. Risset. On the study of VLSI derivation for optical flow estimation. *International Journal of pattern recognition and Artificial Intelligence (IJPRAI)*, 2000.

[46] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in Kahn Process Networks. In *Proceedings of IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, July 17-19 2002.

[47] Alexandru Turjan and Bart Kienhuis. Storage Management in Process Networks using the Lexicographically Maximal Preimage. In *Proceedings of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, June 24-26 2003.

[48] ”www.xilinx.com”. FIFOs Using Virtex-II Block RAM. June 2001.

[49] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. Pngen: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems*, 2007, 2007.

[50] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Realizations of the Extended Linearization Model in the Compaan Tool Chain. In *proceedings of the 2nd Samos workshop*, Samos, Greece, August 2002.

[51] Charles Babbage. *Passages from the life of a Philosopher*. London, 1964.

[52] R.L. Walke, R.W.M. Smith, and G. Lightbody. 20 GFLOPS QR processor on a Xilinx Virtex-E FPGA. In *Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations X*, pages 300 – 310, 2000.

[53] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform–Based Design. *IEEE Trans. on CAD*, 2000.

[54] E. Rijpkema, K. Goossens, A. Radulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, 2003.

[55] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256. ACM Press, 2000.

[56] Tom Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California, Berkeley, CA, December 1995.

[57] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Co., 1986.

[58] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[59] Bart Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[60] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[61] Philippe Clauss and Vincent Loechner. Polylib. http://icps.u-strabg.fr/Polylib, February 2002.

[62] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A Hierarchical Classification Scheme to Derive Interprocess Communication in Process Networks. In *Proceedings of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'04)*, Galveston, Texas, Sept 27-29 2004.

[63] Doran K. Wilde and Sanjay Rajopadhye. Allocating Memory Arrays for Polyhedra. Technical report, INRIA, 1993. Technical Report Number 2059.

[64] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, december 2004.

[65] N. Manjikian and Tarek S. Abdelrahman. Fusion of Loops for Parallelism and Locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[66] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Deriving Efficient Control in Process Networks with Compaan/Laura. *International Journal of Embedded Systems*, 2005. inderscience publishers.

[67] Hristo Nikolov, Todor Stefanov, and Deprettere Ed. Modeling and FPGA implementation of Applications using Parameterized Process Networks with Non-Static Parameters. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005. Submitted for Review.

[68] Tim Harriss, Richard Walke, Bart Kienhuis, and Ed. F. Depettere. Compilation from Matlab to Process Networks Realized in FPGA. In *Proceedings of the 35th Asilomar conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, November 4 – 7 2001.

[69] Jeffrey W. Sheldon, Walter Lee, Benjamin Greenwald, and Saman Amarasinghe. Strength Reduction of Integer Divison and Modulo Operations. In *Languages and Compilers for Parallel Computing*, Cumberland Falls, Kentucky, August 2001.

[70] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[71] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 113, Washington, DC, USA, 2000. IEEE Computer Society.

[72] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated Static Single Assignment. In *IEEE PACT*, pages 245–255, 1999.

[73] J.C.H. Park and M. Schlansker. On Predicated Execution. In *Technical Report HPL-91-58*. HP Labs, 1991.

[74] Greg Snider, Barry Shackleford, and Richard J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 115–124. ACM Press, 2001.

[75] Peter Held. Functional Design of Data-Flow Networks, 1996. PhD thesis, Delft University of Technology, The Netherlands.

[76] Ehud Artzy, James A. Hinds, and Harry J. Saal. A Fast Division Technique for Constant Divisors. *Commun. ACM*, 19(2):98–101, 1976.

[77] Shuo-Yen Robert Li. Fast Constan Division Routines. *IEEE Trans. Computers*, 34(9):866–869, 1985.

[78] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras. Integer Multiplication and Division on the HP Precision Architecture. *IEEE Trans. Computers*, 37(8):980–990, 1988.

[79] T. M. Apostol. *Introduction to Analytic Number Theory*. Springer-Verlag, Berlin, 1975.

[80] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[81] P. Briggs, T. Harvey, and T. Simpson. Static single assignment construction, 1995.

[82] Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 172–181. IEEE Computer Society, 2001.

[83] Navindra Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, February 2006.

[84] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[85] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[86] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[87] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1992.

[88] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), December 1994.

[89] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. In *Proc. 10th International Symposium on Hardware/Software Codesign (CODES'02)*, pages 7–12, Estes Park, Colorado, USA, May 6-8 2002.

[90] C. Leiserson and J. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1, 1983.

[91] T. J. Shepherd and J. G. McWhirter. Systolic Adaptive Beamforming - Radar Array Processing. In *Springer Series in Information Sciences*, volume 25. Springer-Verlang Berlin, 1993.

[92] P. Pango and B. Champagne. Accurate subspace tracking algorithms based on cross-space properties, 1997.

[93] G. W. Stewart. An updating algorithm for subspace tracking. Technical report, College Park, MD, USA, 1990.

[94] A. Kavcic and Bin Yang. A new efficient subspace tracking algorithm based on singular value decomposition. *icassp*, 4:485–488, 1994.

[95] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[96] Franklin T. Luk. A triangular processor array for computing the singular value decomposition. Technical report, Ithaca, NY, USA, 1984.

[97] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Increasing pipelined IP core utilization in Process Networks using Exploration. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'04)*, 2004.

[98] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating Affine Nested-loop Programs to Process Networks. In *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'04)*, Washington D.C., USA, September 23-25 2004.

# Index

Abstract Architecture, 7, 13, 18
address evaluation part, 45
Architecture Synthesis, 7

blocking read, 3
blocking write, 19
bounding box, 45, 57, 58

Communication Synthesis, 8, 39
consumer domain, 17
Control Synthesis, 8, 25

data hazard, 82
Dedicated Channel, 8
dependency graph, 31

evaluation part, 30
Execute Unit, 8, 20
Expression Synthesis, 8
Extended Linearization Model, 44

firing rules, 4

In-Order with multiplicity, 18, 42
In-Order without multiplicity, 18, 42
input arguments, 20
Input Port, 4, 16, 20
Input Port Domain, 20
IP Core Integration, 8, 81
iteration, 15
iteration domain, 15
iterator part, 30

Lexicographical Maximal Preimage, 42
Lexicographical minimal Preimage, 42
LMP evaluation part, 45
local schedule, 17

memory controller, 44
Method of Differences, 69
multiplicity, 17, 41

Network of Synthesizable Processors, 7

Out-of-Order with multiplicity, 18, 42
Out-of-Order without multiplicity, 18, 42
output arguments, 20
Output Port, 4, 16, 20
Output Port Domain, 20

partitioned parameterized predicate controller, 35
PN to Abstract Architecture, 7, 13
Polyhedral Model, 4
Predicated Static Single Assignment, 66, 69
producer domain, 17
producer-consumer pair, 16

rank, 40, 55
read polynomial, 40
Read Unit, 8, 20
reorder memory, 41, 44
reset-$\phi$, 70

selection-$\phi$, 70
selector, 20
self-loop, 5, 53
Semantic Mapping, 8, 13, 18
sequential code, 50

single assignment code, 50
stall, 82
statement, 15
Static Single Assignment, 70
switch, 20

Topological Mapping, 8, 13, 18

variants, 4
Virtual Processor, 8, 19

write polynomial, 40
Write Unit, 8, 20

# Acknowledgment

I would like to thank to Alexandru Turjan and Todor Stefanov for pointing out the importance of a VHDL back-end for the COMPAAN compiler. My thoughts also go to the other members of the LERC group for their kind words and support: Mihai Cristea, Sjoerd Meijer, Vladimir Zivkovic, Jerome Lemaitre, Sylvain Alliot, and Hristo Nikolov. I am grateful to Mingyung Ko, Steven Derrien, Shuvra Bhttacharyya and Shlomo Raikin for the interesting discussions and collaboration. And special thanks to my former professors: Livia Pavelescu, Timotei Ciobanu, Corneliu Popeea, and Adrian Petrescu.

I would like to thank Sebastian Maneth, Dmitry Cheresiz and Ioan Cimpian for the interesting and sometime surrealist discussions we had. I thank Lucian Limona, Laurentiu Nicolae, Nicu Sebe, and Viorel Cristea for their support and friendship, and for being next to me in good times and bad.

To my reviewers, Lucian Limona, Alexandru Turjan and Sven Verdoolaege, sincere thanks for their valuable comments and suggestions, and to Tom de Vries for the Dutch summary of my thesis.

Finally, I would like to thank my family and especially my wife, Daniela, for their support and encouragement. Without them, this dissertation would not have been possible.

# Samenvatting

Sinds de jaren tachtig is er een trend om DSP-functionaliteiten toe te voegen aan processoren voor algemeen gebruik. Tegenwoordig overschrijden de prestatie-eisen van moderne systemen echter vaak het vermogen van deze coprocesoren en zijn er vaak meerdere DSP's nodig. Het opdelen van een DSP-berekening in eenheden die naast elkaar op verschillende hardwarebronnen draaien, is echter een ingewikkeld proces en garandeert op zichzelf geen versnelling. De parallelle code kan zelfs langzamer draaien dan sequentiële software. Voor goede prestaties is het nodig om de communicatie tussen hardware-eenheden te optimaliseren en de gepartitioneerde berekening efficiënt over de hardware-eenheden te verdelen. Een mogelijke oplossing voor het laatste probleem is om bestaande multiprocessorarchitecturen te gebruiken. Deze hebben echter een geheugenhierarchie met meerdere cache-niveaus. Dat maakt de interprocessorcommunicatie zeer gevoelig voor de plek waar de data is opgeslagen. De communicatie voor een DSP-algoritme met slechte datalokaliteit, vertaald naar een dergelijke architectuur, kan veel cycli kosten.

FPGA-platforms bieden steeds meer een hardwarealternatief voor DSP-ontwerpers. Ze combineren alle voordelen van DSP's met bijna de prestatievoordelen van ASIC's. De kracht van FPGAs is dat de ontwerper zo veel parallelle bronnen kunnen gebruiken als het FPGA-platform beschikbaar stelt. De een bepaalde hoeveelheid parallelle bronnen te gebruiken komt de vereiste prestaties binnen bereik. Bovendien is het mogelijk een specifiek afgestemde communicatiestructuur te definiëren en daarmee de complexiteit van een moderne processorgeheugenhierarchie te vermijden.

De DSP-architectuur voor algemene doeleinden is optimaal voor toepassingen die sequentieel draaien en een groot globaal geheugen gebruiken. Het FPGA-platform gaat echter uit van parallelle logica en gedistribueerd geheugen. De voornaamste vraag is daarom hoe je overstapt van een sequentiële, globale geheugenspecificatie naar een parallelle, gedistribueerde geheugenarchitectuur. Handmatige overzetten is vaak een moeizaam proces waarbij gemakkelijk fouten kunnen optreden. We zouden deze overstap dan ook willen automatiseren.

Er zijn twee manieren waarop we het automatiseren kunnen oplossen. Een manier is om applicatie ontwikkelaars hun applicaties in een parallelle programmeertaal (textueel of grafisch) te laten ontwikkelen. Grafische of visuele programmeerstijlen zijn voorgesteld en gebruikt om signaal processing en multimedia applicaties te specificeren. Typische voorbeelden van zulke parallelle programmeerstijlen zijn gebaseerd op dataflow grafen en dataflow proces netwerk berekeningsmodellen. In deze modellen bestaat een programma uit actieve entiteiten (functies, threads, processen) die point-to-point communiceren over FIFO kanalen. Applicatie ontwikkelaars zijn terughoudend in het specificeren in termen van deze modellen om verschillende redenen. Allereerst zijn de modellen niet expressief genoeg, of onbeslisbaar. Ten tweede, praktische applicaties kunnen niet gespecificeerd worden in termen van dataflow modellen die niet echt rekening houden met dynamische control flow.

De andere manier om de mismatch tussen sequentiële imperatieve applicatie specificatie en het parallelle executie platform op te lossen, is het converteren (paralleliseren) van de sequentiële specificatie naar een invoer-uitvoer equivalent parallelle specificatie die beter aansluit op een multi-processor executie platform. De geparalleliseerde code is dan afgebeeld op het multi-processor executie platform. Niet elk programma in een sequentiële imperatieve taal kan makkelijker - of zelfs automatisch - geconverteerd worden naar een invoer-uitvoer equivalent parallelle specificatie. Echter, in signaalverwerking, multimedia, moleculaire biologie, en andere gerelateerde applicatie domeinen, zijn er veel programma's die geconverteerd kunnen worden naar een invoer-uitvoer equivalent parallelle specificatie. Deze klasse van programma's word *nested loop programs* (NLP) genoemd. In het bijzonder kunnen de zogenaamde affine nested loop programs automatisch geconverteerd worden. Meer specifiek, in deze thesis behandelen we alleen *static affine nested loop programs*. We adresseren het probleem van het synthetiseren van *Process Network* specificaties naar FPGA multi-processor executie platformen. De proces netwerken die we behandelen zijn speciale gevallen van Kahn Process Networks. We noemen deze dan ook COMPAAN Data Flow Process Networks (CDFPN) omdat deze gegenereerd worden door een vertaler genaamd COMPAAN, die automatisch affine nested loop programma's vertaald naar invoer-uitvoer equivalent parallelle specificaties. Ons doel in deze thesis is om een effectieve en efficiënte implementatie van CDFPNs in een FPGA executie platform te leveren, waarbij onze implementatie bijna een één-op-één afbeelding van het originele CDFPN is. Het executie platform ontstaat uit het afbeelden van een gegeven CDFPN specificatie tot een specifiek multi-processor executie platform op een FPGA. In de afbeelding gebruiken we nog geen embedded CPU blokken, gespecialiseerde DSP blokken of soft-core processoren, hetgeen een natuurlijk uitbereiding zou zijn van het werk gepresenteerd in deze thesis.

# Curriculum Vitae

Claudiu Zissulescu was born on December 21, 1976 in Bucharest, Romania. In 1995 he received his Baccalaureate from the ”Octav Onicescu” High School in Bucharest. In 2000 he graduated from ”Politehnica” University of Bucharest with a Dipl. Ing. and M.Sc. degree in Computer Science. In the same year he joined the Leiden Embedded Research Center (LERC), a part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, where he was appointed as a research assistant (AIO). During his work within LERC, Claudiu has carried out research on the mapping of stream-oriented media applications onto parallel architectures.

Upon completing his research, he joined Chess BV in Haarlem, The Netherlands, where he worked on the development of a wireless sensor network. In 2008, Claudiu has joined DSP Innovation Center at NXP Semiconductors where he currently works on the design of next-generation platforms for wireless communication.

ISBN/EAN 978-90-9023643-8