Universiteit
Leiden
The Netherlands

Compiling nested loop programs to process networks

Turjan, A.

Citation
Turjan, A. (2007, March 22). *Compiling nested loop programs to process networks*. Retrieved from https://hdl.handle.net/1887/11454

Version:	Not Applicable (or Unknown)
License:	Leiden University Non-exclusive license
Downloaded from:	https://hdl.handle.net/1887/11454

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

Compiling Nested Loop Programs to Process Networks

Alexandru Turjan

Compiling Nested Loop Programs to Process Networks

Proefschrift

ter verkrijging van de graad van Doctor aan de Universiteit Leiden, op gezag van de Prof.Mr.Dr. Paul F. van der Heijden, volgens besluit van het College voor Promoties te verdedigen op dinsdag 22-03-2007 klokke 15:00 uur

door

Alexandru Turjan
geboren te Pitesti, Romania
in 1977

Samenstelling promotiecommissie:

promotor	Prof.dr. Ed F. Deprettere	
co-promotor	Dr. Bart Kienhuis	
referent	Dr. Erwin de Kock	NXP Semiconductors, Eindhoven
overige leden:	Dr. Jos van Eijndhoven	NXP Semiconductors, Eindhoven
	Prof.dr. Harry Wijshoff	
	Prof.dr. Chris Jesshope	Universiteit van Amsterdam
	Prof.dr. Sjoerd Verduyn Lunel	

Compiling Nested Loop Programs to Process Networks
Alexandru Turjan. -
Thesis Universiteit Leiden. - With index, ref. - With summary in Dutch
ISBN 978-90-9021676-8

Copyright ©2007 by Alexandru Turjan, Eindhoven, 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

parintilor si fratelui meu: Cecilia-Aurelia, Dante si Mihail

Contents

Contents vii

1 Introduction 1

1.1 Problem Definition . . . 3
1.2 Analysis framework . . . 4
1.3 Solution approach . . . 6
1.4 Contributions . . . 8
1.5 Related Work . . . 8
1.6 Thesis Outline . . . 10

2 Background 11

2.1 Polyhedral Model . . . 11
2.2 Matrix decompositions . . . 13
2.2.1 Smith Normal Form (SNF) . . . 13
2.2.2 Hermite Normal Form (HNF) . . . 13
2.3 Parametric Integer Linear Programming . . . 14
2.3.1 Linear Programming (LP) . . . 14
2.3.2 Integer Linear Programming with Gomory cuts . . . 15
2.3.3 Parametric programming . . . 16
2.3.4 Empty Domain Test . . . 16

3 Consumption Restructuring 17

3.1 Problem Definition . . . 17
3.2 Solution - ILP approach . . . 18

3.3 An alternative approach . . . 21
3.4 Discussion . . . 22

4 Production Restructuring **23**

4.1 Problem Definition . . . 23
4.1.1 Formal representation . . . 24
4.2 Solution - ILP approach . . . 25
4.3 An alternative solution - the SNF approach . . . 27
4.4 Putting things together - a hybrid approach . . . 30
4.4.1 The SNF & HNF guarded with ILP approach . . . 30
4.4.2 Restructuring the hybrid approach . . . 34
4.5 Discussion . . . 40

5 Communication Model Selection **41**

5.1 Problem Definition . . . 42
5.2 Solution - ILP approach . . . 43
5.2.1 The Multiplicity Test . . . 43
5.2.2 Optimizations . . . 46
5.2.3 The Reordering Test . . . 49
5.2.4 Optimization . . . 50
5.3 Experimental Results . . . 53
5.4 Alternative Polynomial-time Classification Techniques . . . 53
5.5 Order and Multiplicity detection using the P/C mapping matrix . . . 54
5.5.1 Order . . . 55
5.5.2 Multiplicity . . . 56
5.5.3 An In-order example . . . 56
5.5.4 An Out-of-order example . . . 57
5.6 Lineality space . . . 57
5.6.1 Ordering and multiplicity characteristics in the presence of a Consumer lineality space . . . 58
5.6.2 Example - Reordering in case of a Consumer lineality space . . . 61
5.6.3 Example - Multiplicity . . . 62

5.7 Limitations . . . 63
5.8 Putting things together - a hybrid approach . . . 64
5.9 Results . . . 65
5.10 Discussion and conclusions . . . 66

6 Process Network Synthesis **67**
6.1 Problem Definition . . . 68
6.2 Solution . . . 68
6.2.1 Removing the global schedule . . . 71
6.3 Converting IPN to PN . . . 74
6.3.1 Process local schedule . . . 74
6.3.2 Synchronization policy / Life-time analysis . . . 74
6.3.3 Communication synthesis / Address generation . . . 93
6.4 Running example . . . 100

7 Network Transformations **103**
7.1 Process Splitting . . . 104
7.1.1 Solution . . . 105
7.1.2 Example - splitting without multiplicity . . . 110
7.1.3 Example - splitting with multiplicity . . . 112
7.2 Channel Merging . . . 114
7.2.1 Basic Terminology . . . 115
7.2.2 Problem Statement . . . 116
7.2.3 Empty Domain Test . . . 118
7.2.4 Basic FIFO Merging Test . . . 118
7.2.5 Extending the Merge Test to more than two FIFO channels . . . 121
7.2.6 Implementation and Results . . . 122

8 Conclusions and Future Research **123**
8.1 Compilation Flow Overview and Results . . . 123
8.1.1 Experimental Results . . . 124
8.2 Directions for Further Research . . . 125

8.2.1 Source code transformations 125
8.2.2 Memory estimates 126

Appendix: Memory estimates **127**

Bibliography **133**

Index **141**

Acknowledgments **143**

Samenvatting **145**

Chapter 1

Introduction

Applications envisioned for the next decade in the area of image, (cellular and radar) signal processing or bio-informatics have a high computational demand. During the last decades, the most popular approach to satisfy the applications computational demand was to provide (faster) processors that run at higher frequencies. This was a result of the fact that the number of transistors on a chip was doubling about every two years (this observation is well known as Moore's law); smaller transistor designs allowing faster clock rates. However, due to technological constraints such as wire delay and power leakage (the Moore's law reaches its limits), smaller transistors don't necessarily run any faster. To solve this problem, the emerging hardware platforms make use of certain application's characteristics (e.g., thread and data level parallelism) by exploiting physical on-chip locality (to keep wires short). Those platforms, referred to as *heterogeneous multiprocessor platforms* [45], are typically composed of loosely coupled compute cores that exchange data using programmable interconnections such as a network on chip or a switch matrix [34, 80]. The compute cores can be programmable (CPUs or DSPs) or specialized IPs (ASIC or FPGA). The reasons of using different cores are due to power consumptions, data throughput and cost considerations. Depending on the partitioning of an application, the programmable cores are usually used to implement the low to medium throughput part of the application, while the specialized IP cores are used to implement the high throughput part of the application. Using different compute cores for different parts of an application, gives the flexibility to the system designer to choose for an effective hardware platform implementation.

As a result of the advancement in VLSI technology that allows higher and higher transistor integration, heterogeneous multiprocessor platforms can be integrated in a single chip [72, 108, 85]. However, the platforms must be programmed/reconfigured and therefore, dedicated software/bit-streams must be developed to obtain the desired performance. It is generally accepted that in order to cope with the multiprocessor characteristic of the new hardware platforms [84], a parallel *model of computation (MoC)* should be involved as programming model. In case of streaming applications the *Process Network (PN)* model of computation [51, 60] is a fair choice. The PN is a deterministic parallel MoC that explicitly specifies tasks as processes and distributed memory as FIFO channels. The localized control and distributed memory in a PN are the two key ingredients that allow to program heterogeneous multiprocessor platforms. The localized control matches the loosely coupled platform hardware components and the distributed memory matches the style of interaction between the components.

Software development for a given platform is a complex and time consuming job, which becomes even more difficult when dealing with the details of parallelism and memory management [35] as in the case of a PN representation. According to [78] there are three possibilities to design a parallel program:

- write a new parallel program,
- adapt an already existent parallel program which solves a similar problem, or
- detect and exploit inherent parallelism present in an already existing sequential program.

The first option relies on the ability of a system designer to manually partition the application's core computation, memory and control across the platform hardware components [31]. This process is typically performed in an empirical manner, lacking a systematic solution approach. Writing parallel code involves all the issues of the sequential style of writing code plus many other issues regarding the extraction of application component tasks, the synchronization between the tasks, and the memory management. Such job is time consuming which conflicts with the decrease of the time-to-market of nowadays embedded products.

The second option is not suitable to cope with embedded systems rapid market growth in which more and more applications and new standards are designed and embedded in mass-market products. To adapt an existent parallel program would require support from a parallel debugger and profiler. However, such tools are not mature enough at this moment requiring future research and development.

According to the third option, parallelism is automatically exploited by a compiler. In this case, based on dependence analysis, a parallelizing compiler performs a suite of transformations on a sequential program in order to derive a parallel program. Thus, by means of a parallelizing compiler the designer does not have to explicitly specify parallelism using special language constructs or compiler directives. The underlying system is transparent to the user who doesn't have to care about the complexity of parallelism, delegating in this way the hard task to the compiler designer [35]. This approach is interesting for the following reasons:

- the programming style is widely accepted as it relies on the von Neumann paradigm, i.e., the programs are specified as a number of tasks to be performed in sequential order by a single processor.
- the programmer (almost) does not need to know about parallel programming or parallel architectures to exploit inherent parallelism in the application,
- by choosing a different back-end, the compiler can relatively easy, retarget new architectures, and
- there exists powerful support for developing sequential applications like integrated software development environments that include compiler, debugger and profiler.

However, there are many who believe that the parallelizing approach is not a solution and that explicit parallelism is needed. Their belief is based on the performance achieved by the current parallelizing compilers, and partially by the theorem of Bernstein [10], which states that:

Theorem 1 *In general it is undecidable whether two arbitrary operations in an imperative sequential program can be executed in parallel.*

According to this theorem, there is no automatic compile or run-time support that can fully exploit all potential parallelism in a sequential application. A solution to overcome this theoretical limitation is to make use of a sequential programming language which makes parallelism extraction possible. As presented in this thesis, we consider for compilation a subset of an existing programming language. In this context we have built the *Compaan compiler* [58] that compiles static affine nested-loop programs

specified in a sequential language like MATLAB into functionally equivalent parallel representations namely Process Networks.

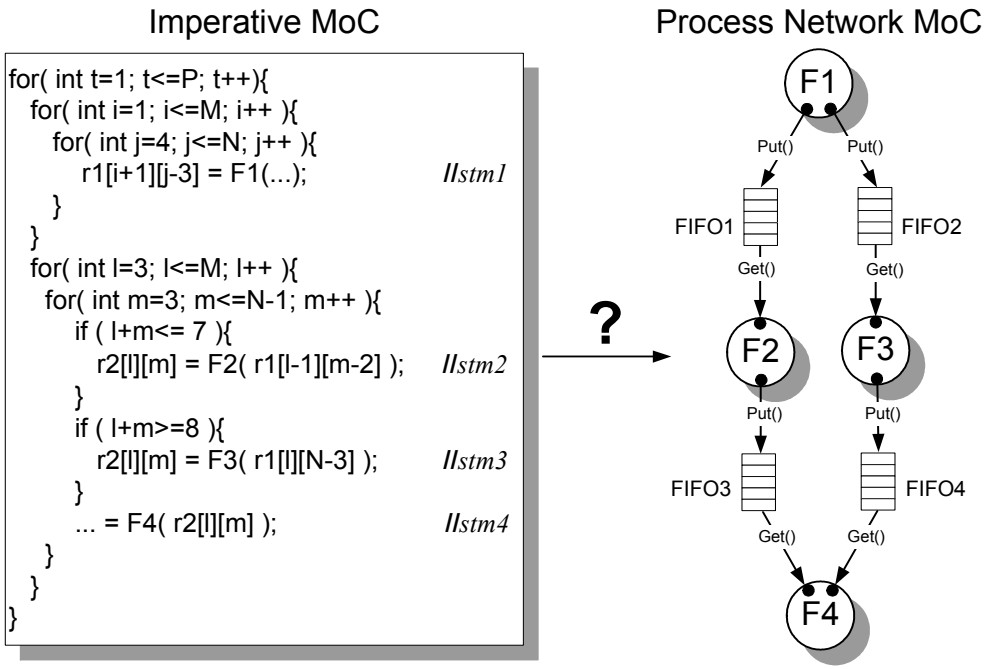


Figure 1.1: How to compile an affine nested loop application into a Process Network?

1.1 Problem Definition

Currently, the most popular programming languages remain sequential in nature, requiring the use of advanced compilers to allow their mapping onto parallel architectures. In this thesis we address the problem of converting an imperative sequential specification to an equivalent dataflow specification, namely a process network. An example of such conversion is shown in Figure 1.1. Bellow we describe two differences between the imperative and the dataflow models of computation and show their implications:

- In the imperative MoC the statements are executed sequentially, one at a time, and their execution order is specified explicitly in the program. We remark that in many cases completely sequential execution of the statements imposed by use of the imperative MoC can be relaxed without compromising the correctness of computation. Consider the example given in Figure 1.1. For the outer loop iterator value $t = 1$, the statement *stm1* is executed $(M \times N)$ times followed then by the execution of statement *stm2*. However, *stm2* can be executed immediately after the first execution of *stm1* without having to wait $(M \times N - 1)$ further executions. A PN model depicted in the right part of the figure, specifies a computation equivalent to the one on the left side but without unnecessary statement ordering restrictions. In a PN the statements are executed asynchronously as soon as the statement operands are available. In this way more than one statement can be executed at a time.

- In the imperative MoC, when the same program has to be executed on a new set of input data, it has to wait until the previous instance of the program has completed. In the example given in Figure 1.1, executing two instances of the code fragment on the left-hand side would corrupt the memory (array A[]), leading to erroneous computation. In case of a PN MoC, however, new input data can be fed into the network as soon as it becomes available, without waiting for completion of the computations on the previous data. This makes PN a model of computation suitable for expressing so-called *streaming computations*.

The main difference between the two models of computation is related to the ordering constraints among their program statements. Therefore, to synthesize a PN starting from a sequential specification we have to be able to analyze the ordering constraints among the program statements. Essentially, those ordering constraints are dictated by the data dependence relations existent within the program statements. Those dependences together with the program statements form the program *Data Dependence Graph (DDG)*, where the statements represent the nodes of the DDG and the dependences represent its edges. The order of the program statements can be rearranged without changing the program functionality under the constraints imposed by the edges of the DDG. Based on the DDG we can formulate the conversion from a sequential program to a PN as a graph manipulation problem. The work presented in this thesis investigates an efficient and systematic method for collapsing the nodes of a DDG into a number of autonomously running processes, while grouping its edges over a reduced number of communication channels allowing the synthesis of a synchronization policy among the processes.

1.2 Analysis framework

In this section we give a high level presentation of the analysis used during the compilation of a sequential program to a PN. We know that data dependence relations are used to capture the ordering constraints of the program statements. A program statement may dependent on another statement because both statements share a common memory location, i.e., *data dependence*. As already discussed, we need to manipulate the ordering of the statements of the input program. However, the ordering of the statements has to respect the data dependences existent between the program statements. To capture the data dependences, the program DDG is built by performing *array data flow analysis* [38, 75, 66, 56, 6, 107]. Each node of the DDG represents an assignment statement and the edges connecting the nodes represent the data dependences existent between the program statements. The data dependences can be of one of the following four types: flow, anti, output or input dependency. To define those types of dependences we use of the following two sets:

- $IN(S)$ is the set of memory locations that are read by statement S and
- $OUT(S)$ is the set of memory locations that are written by statement S.

Assume that within a sequential program statement A is executed before statement B. Thus, between statement A and statement B may exist data dependences as shown in Table 1.1.

According to Bernstein [10] and Kuck [59] the program statements can be reordered (executed following a different order) as long as the flow, anti and output dependences are respected. To derive a PN from a sequential specification we have to manipulate the sequential specification under the constraints imposed by those types of dependences. Although two input dependent statements can

OUT(A) ∩ IN(B) ≠ ∅	$A\ \delta^f\ B$	flow (true) dependence
IN(A) ∩ OUT(B) ≠ ∅	$A\ \delta^a\ B$	anti dependence
OUT(A) ∩ OUT(B) ≠ ∅	$A\ \delta^o\ B$	output dependence
IN(A) ∩ IN(B) ≠ ∅	$A\ \delta^I\ B$	input dependence

Table 1.1: Data dependences types

exchange their order of execution without any impact on the behavior of the program, to support their parallel execution the target platform should provide multiport memories. This type of memory is expensive and limited from the point of view of parallel accesses. Therefore, we manipulate the input dependences by performing certain program transformations that avoid the need for multiport memories. Similarly we will perform transformations to remove some of the anti and output dependences to allow more parallelism to take place.

The program dependences form the program dependence graph, which sometimes is called the precedence graph. The vertices of the graph represent statements in the program (source-level statements, intermediate language statements, basic blocks, or assembly instructions), and the edges represent dependences. In Figure 1.2 we show a piece of program together with the corresponding DDG.

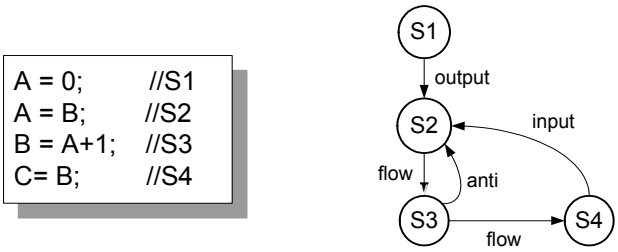


Figure 1.2: Example of a DDG

Due to the incapability of analyzing some of the relations[1] between the statements of a program, the corresponding DDG can be inaccurate in the sense that it may contain more edges (dependences) than necessary (e.g. as when performing address-based dependence analysis - see page 138 in [107] - which for the program given in Figure 1.2 leads to an extra flow dependency between statements $S1$ and $S3$). To allow statement reordering, it is essential to have an exact and accurate DDG representation. Deriving an accurate representation of the DDG is in general a hard task that requires the analysis of the data (memory) accesses under the control constraints imposed by the conditionals of the program. Moreover a program may store data into memory arrays, may contain data dependent control statements, and nested loops which make the establishment of the DDG even harder. In such case less accurate analysis is performed such that the DDG may include extra edges. In such case the analysis of the DDG is less permissive and makes it a harder task for a parallelizing compiler when trying to reorder program statements.

The class of applications we consider for compilation to a PN representation, is restricted to nested loops with static control and affine indexes [38]. The restriction imposed here refers to the largest class of programs onto which exact dependence analysis can be performed. Due to this analysis, the DDG of the application can be established exactly (i.e., exact specification of the program dependences)

[1] As caused by data-dependent control conditions

at compile time. As we are dealing with nested loop programs the dependence analysis that we use establishes the dependence relations between instances of the program assignment statements by indexing (superscripting) a statement with the loop iterators. In loops, each statement may be executed for a number of times. To represent each statement execution individually is unfeasible and sometimes even impossible (e.g., a parametric loop bound condition) and, therefore, to represent the data dependences we make use of an abstraction of the DDG, called reduced dependence graph (RDG) [26]. Due to the restriction of our class of input applications, we can derive for each statement execution a dependence distance that is a vector pointing to the statement that produces data to the considered statement.

An example of such an input application is given at the left side of Figure 1.1, where each assignment statement is iterated over a convex domain called *iteration space (IS)* composed of *iteration points (IPs)* [5]. The iteration spaces can be parameterized by using for-loops with parametric bounds as can be observed in the code. The PN that is generated consists of a number of processes, each process executing one of the input program assignment statements for a number of times. For example, process F1 corresponds to statement $stm1$, process F2 to statement $stm2$, and so on. In the compilation from an affine nested loop program to a PN, two problems arise. First, the computation carried out by a sequential application in a single thread process needs to be distributed over a number of separate computational processes, each one corresponding to a different thread of control. Second, the global memory arrays (e.g., $r1$ and $r2$) used for data storage need to be transformed to dedicated communication channels that are accessed using a blocking read and write primitives, providing in this way a simple inter-process (thread) synchronization mechanism.

1.3 Solution approach

As shown in Figure 1.3, the conversion of a sequential program to a PN takes place gradually in a number of steps which are mainly dealing with: a. *the extraction of independent memory accesses* and b. *the conversion of the dataflow into managed data movement over a well defined communication structure*. The conversion methodology is based on the *polyhedral model*, *array dataflow analysis*, and *integer linear programming (ILP)*. A basic understanding of these concepts and techniques will allow a fast understanding of the thesis. The most relevant background concept is the array dataflow analysis done by Paul Feautrier [38].

As a result of a *Preprocessing* step, the input program abstract syntax tree is converted to a compiler internal network structure where all the executions (iteration space) of one assignment statement are collapsed into a separate structure called process. The network structure is based on the polyhedral model and uses an algebraic representation of the input program. The reason to use this structure is that the abstract syntax tree is not appropriate for complex program restructuring that modifies, for example, the order of executing the program statements. The topology of this network resembles the RDG of the program given in the left part of Figure 1.1. Each circle from the left part of Figure 1.3 represents a process iterating one of the assignment statements over the same iteration space as the statement is iterated in the original code. However, these processes are still executed one at the time following the global schedule in which the corresponding assignment statements are executed in the original code. This network structure represents the input of the *Consumption Restructuring* step. In this first step, we restructure the data consumption, i.e., basically each array used for storing data generated by different producer processes is replaced by a number of separated memory

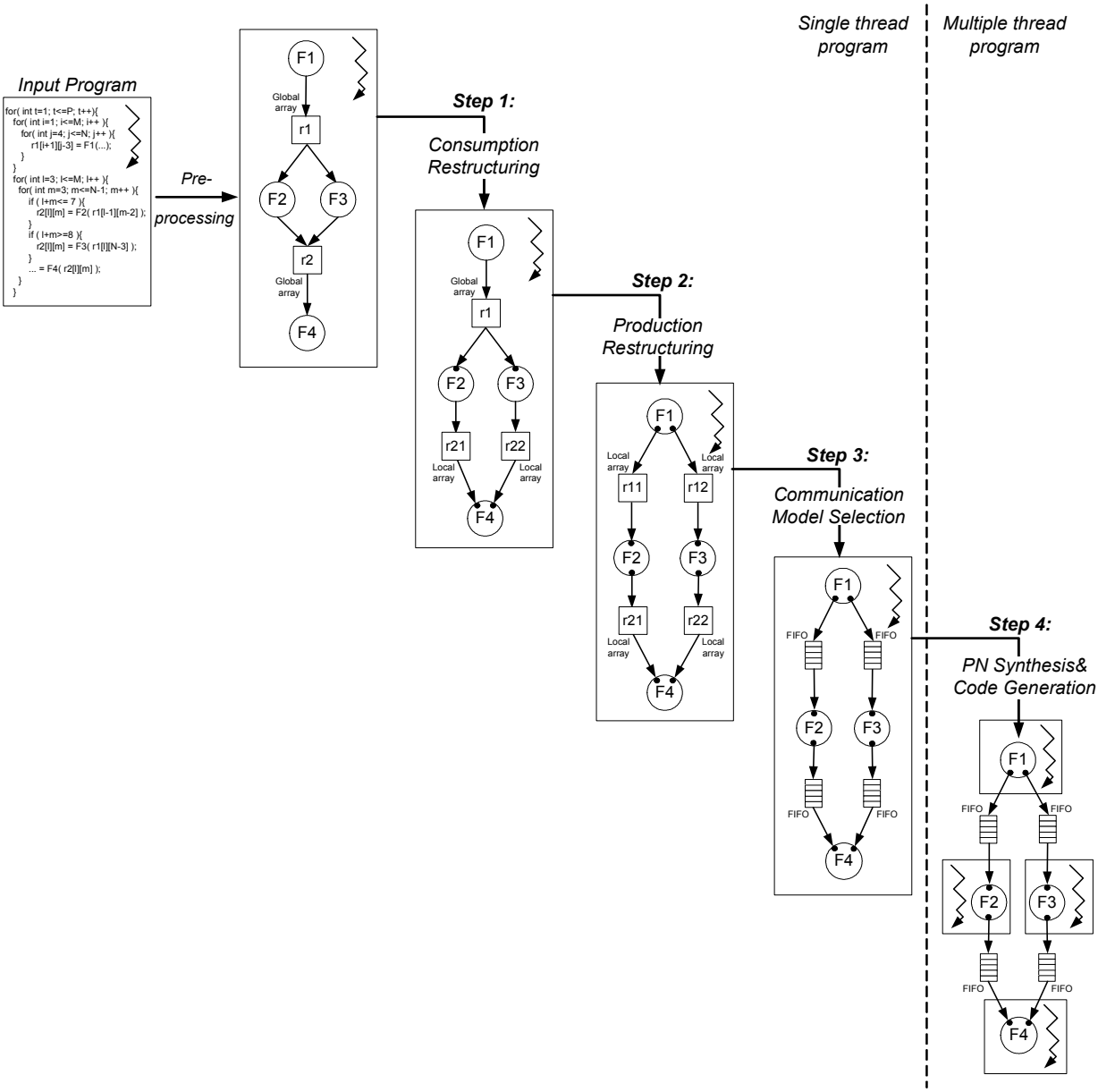


Figure 1.3: Deriving a Process Networks in four steps

arrays, one for each producer process. In the second step, *Production Restructuring*, we restructure the data production, i.e., basically each array used for storing data consumed by several consumer processes is replaced by a number of separated memory arrays; one for each consumer process. After performing these two steps, a distinct piece of memory is used to communicated data between each producer and consumer process. This forms an instance of the classical *producer/consumer (P/C) pair*. Fast P/C communication and synchronization mechanisms are key elements in synthesizing a high throughput PN. Depending on the order the data is produced and consumed in a P/C pair, specialized communication mechanisms with adequate inter-process synchronization policies have to be established. This is done in the third step, called *Communication Model Selection*. Using the

information obtained in first three steps, a PN with autonomously running processes communicating data over FIFO channels is obtained during the *Process Network Synthesis* step. Based on different visitors this network can be instantiated as C++ or Java code making calls to a multi-threaded library [30, 28]. Because there is a direct correspondence between the process network components and the FPGA hardware components, it turns out that with some extra effort the same process networks can be efficiently (in terms of data throughput and resources) mapped onto FPGA platforms. This is achieved under the assumption that each of the input program assignment statements represents a complex computational kernels. Hence, by retargeting the compiler it becomes possible to obtain a hardware implementation of the PNs using the *Laura* [113, 111, 110, 112] VHDL back-end.

1.4 Contributions

In the context of the work done by Deprettere et al. in the Hifi project, parallelization of nested loops with static control and affine indexes was addressed by Held [49] who tried to automatically generate systolic arrays by using data-flow analysis and dependence graphs. Further on, inspired by this work, Rijpkema [79] stated the problem of translating the class of nested loops to process networks. In [58], he presented an approach for conducting the translation in a number of steps. In some of these steps, he relied on the Ehrhart theory [33, 21] which, due to its computational complexity, has implementation limitations [100] and provided unfeasible expensive address generation used for communicating data between processes. Furthermore, this theory is applicable only in the context of a limited number of applications[2] which in Rijpkema's approach were not detectable at compile time in order to guarantee a correct PN. To overcome these limitations, we present in this thesis a new compilation procedure. The main contributions of the thesis can be summarized as follows:

- The thesis presents a novel, fully analytic procedure to compile any nested loop program with static control and affine indexes to a functionally equivalent process network [95].
- The compilation procedure uses a number of novel ILP formulations [91, 94] that in general can be solved with an adequate ILP solver [37, 73], more stable than the Ehrhart support provided by PolyLib [105]. However, for all the steps alternative solutions which do not require the ILP solver are presented as well [93, 96].
- All steps have been implemented in the Compaan compiler, replacing and extending the steps presented in [58], making the compiler robust and capable of converting *any* nested loop programs with static control and affine indexes to process networks [97].
- The compilation procedure was evaluated on a number of image and signal processing applications showing that the approach is capable of automatically generating efficient process networks in acceptable computation time [84, 109, 68, 87].

1.5 Related Work

A tremendous amount of work has been done over the last 25 years to develop compilers that enable sequential programs to execute efficiently on parallel architectures. General information on this subject

[2] Namely loop nests that involve for manipulation iteration spaces (see Section 1.1) described by polytopes and not linearly bounded lattices [89].

can be found in [107, 56, 26, 6]. It seams that the effort was successful in the area of vectorizing compilers that enable running code on vector architectures. However, parallelizing compilers capable of generating code for running onto multiprocessor architectures seam to be less successful. The main reason of this inefficiency is that they aim to identify parallel loops which allow to construct parallel threads i.e., threads that run without interfering with each other. In the context of modern applications where most of the program assignments are interdependent, this approach appears to be rather restrictive. Thus, in this thesis we address this limitation by extracting threads together with adequate synchronization policies. As further presented, there are a number of research projects dealing with the automation of mapping sequential applications onto multiprocessor platforms. Compared to our project those projects rely on similar compiler techniques e.g., data dependence analysis, control flow analysis and loop transformations. However, one of the main differences is that our compiler outputs an asynchronous program representation while most of those projects derive scheduled representations.

The Intel Hyper-Threading Technology processors can significantly increase the performance of application programs with a high degree of parallelism. These potential performance gains are obtained if an application is efficiently multi-threaded, by manual or automatic parallelization techniques. The Intel C++/Fortran compiler supports automatic loop parallelization [12, 90] by building parallel threads out of the outer-most sequential loops. The threads are generated by making use of a multi-threaded run-time library. Compared to our approach their parallelization is more restrictive as it applies only to loops without carried dependences detected and manipulated using classical data dependence tests. Another difference is that the Intel threading parallelization technique can be combined with the vectorization of the of innermost loops [11]. In the context of our work, vectorization is an interesting future research topic briefly discussed in Section 8.2.1.

The PICO project [54, 86] at HP Labs 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 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.

The Phideo design methodology [103, 102] uses a tool set that inputs a sequential specification and outputs a data-flow hardware structure consisting of processing units, memories to communicate data between the processing units, address generators used to address the memories and a controller used to synchronize the processing units. Thus, the hardware designed by Phideo is synchronous and a global schedule over each component block 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.

Another example is the Atomium [18] 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 one, 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.

1.6 Thesis Outline

The compilation procedure presented in Figure 1.3 consists of a number of steps and we present each step by describing first the main idea in an intuitive way, followed by the translation of the idea into its ILP formulation, then an alternative non-ILP solution is given when possible and finally, how the idea works out on a running example. Overall, the thesis is organized as follows:

In Chapter 2, we describe the mathematical model (used by the compiler as internal representation) together with the main techniques and tools used throughout the thesis to formulate and solve the steps of the presented compilation approach.

In Chapter 3, we describe in detail the Consumption Restructuring step. During this step the allocation scheme concerning the memory accesses used for storing data is changed such that each producer assignment statement will store data into a separate memory array. This transformation is realized by breaking all of the anti dependences and some of the output dependences present in the input program (namely those with respect to different assignment statements).

In Chapter 4, we describe the Production Restructuring step. During this step the allocation scheme concerning the memory accesses used for reading data is changed such that each consumer assignment statement will read each input data from a separate memory array. This transformation is realized by breaking the program input dependences with respect to different assignment statements.

In Chapter 5, we describe the Communication Model Selection step. This step presents a classification scheme used to recognize the communication pattern of an arbitrary producer/consumer pair of assignment statements. This analysis makes possible the conversion to a parallel representation of the sequential program obtained so far.

In Chapter 6, we describe the Network Synthesis step during which the sequential program obtained after the first two compilation steps is converted into a process network in which the processes run asynchronously without following a global schedule over the whole network. We also analyze efficient realizations of the communication structure used for each type of communication channel.

In Chapter 7, we present a set of two transformations, performed at the network level in order to improve the parallelism and the memory usage. The first transformation deals with an automatic splitting of the network's processes in order to better exploit the application data-parallelism. The second transformation performs a communication network restructuring by merging some of the network communication channels.

Finally in Chapter 8, we conclude the work by presenting the Compaan compiler organization and show results from passing through out compiler a number of classical applications. We end up the chapter by presenting three future research ideas to be investigated in this thesis context.

Chapter 2

Background

2.1 Polyhedral Model

The traditional compiler internal representations are abstract syntax trees (AST) and lists of three address statements [1]. However, in the context of loop transformations those representations are difficult to manipulate and involve complex operations. To address this problem, a special intermediate program format called *Polyhedral Model* originally introduced for systolic array synthesis [53] was found to be suitable for parallelizing compilers [38]. This model applies to the class of programs which we address for compilation in this thesis, i.e., *static affine nested loop programs (SANLP)*. As suggested by the name, these programs are characterized by the following properties (see the program at the left side of Figure 2.1):

- control statements are for loops with affine conditional bounds and constant step size, and if conditionals with affine conditions,
- the memory array subscripts and conditionals are affine constructs of outer loop counters, scalars which do not change during program execution, and constants,
- all the data dependences existent between two different assignment statements are exposed via their input and output parameters. Remark here that the restriction does not refer to execution of the same assignment statement.

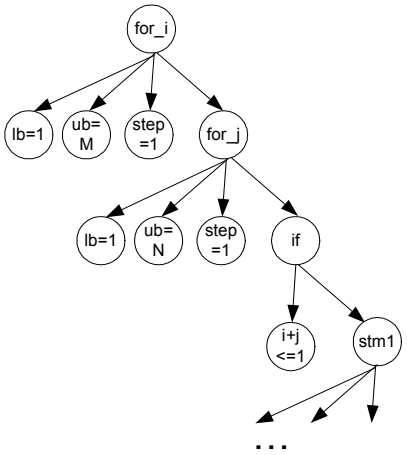
The polyhedral model relies on the polyhedral theory and the combinatorial optimizations theory as support to represent and perform program transformations. To support the automation of program transformations a number of libraries like PPL [4], Polka [50], Porta [19], PolyLib [22], PIP [37], Omega [73], Cdd [39], Polymake [40], lrs [3] and qhull [8] have been developed. In our compiler we make use of PIP or Omega and PolyLib.

In Figure 2.1 we give a SANLP for which the corresponding AST and the geometrical & algebraic polyhedral representations are provided. Using the iterators of the loops surrounding a program assignment statement, each execution of the statement $stm1$ is represented by an iteration vector $v_{stm1} = (i_1, i_2, ..., i_n)$. Using the iteration vector v_{stm1} the whole program execution of statement $stm1$ is described by the following constructs:

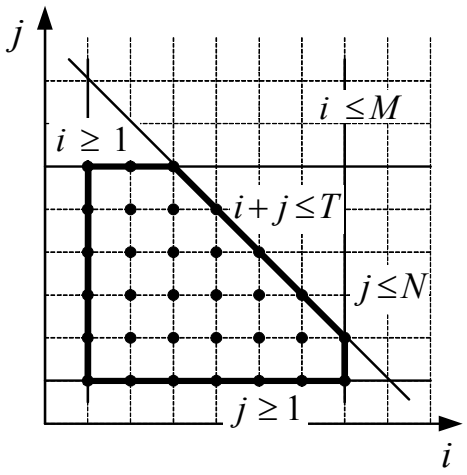
a. Nested loop program:

```
for( int i=1; i<=M; i++ ){
  for( int j=1; j<=N; j++ ){
    if( i+j<=T ){
      r1[i+1][j-3] = F1(...);   //stm1
    }
  }
}
```

b. AST:



c. Geometrical representation of the *stm1* iteration space:



d. Algebraic representation of the *stm1* iteration space:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 0 \\ -1 & 0 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{bmatrix} 1 \\ -M \\ 1 \\ -N \\ -T \end{bmatrix}$$

Figure 2.1: Polyhedral representations of a SANLP

- *iteration space (IS)*: The iteration space (or *iteration domain*) IS_{stm1} represents the set of values of the iteration vector for which statement $stm1$ is executed. Because the statement execution is guarded by affine control, its iteration space IS_{stm1} can be specified by a set of linear inequalities defining a Z-polyhedron[1] $IS_{stm1} = \{v \in Z^n \mid Av \geq c\}$, where v is the statement iteration vector, A is an integral matrix and c is the constant vector. The constant vector c may contain parameters like in Figure 2.1 where $c = [1, -M, 1, -N, -T]^T$. In this case the Z-polyhedron describing the iteration space is parametric.

- *execution order*: An affine function $O(v) = Sv + t$ specifying for each iteration point in the iteration domain an execution order. Namely the execution of v_1 takes place before the execution of v_2 iff $O(v_1) < O(v_2)$. For further details of how this function can be established we refer to [37]. The order of an iteration point can also be established based on the *lexicographic order*: $v_1 = (i_1, i_2, ..., i_n) \prec v_2 = (j_1, j_2, ..., j_n) \equiv \exists k \; s.t. \; \forall \; s < k : \; i_k = j_k \vee i_s = j_s$, where $''\prec''$ denotes the lexicographic order. However, the lexicographic order is not enough to induce a total order $''\prec_t''$ over all the statements of a program e.g., two statements with the same execution vector can not be ordered solely using the lexicographic order. To solve this problem we are using the following boolean function: $T(stm_1, stm_2) = true$ when stm_1 precedes textually statement stm_2, else T is false. Based on this function, a total order among

[1] A Z-polyhedron is the set of integer solutions to a system of linear inequalities.

the program statements follows:

$$stm_1(v_1) \prec_t stm_2(v_2) \equiv v_1 \prec v_2 \vee T(stm_1, stm_2) = 1. \quad (2.1)$$

In the polyhedral context we make use of:

- basic polyhedral manipulations: intersection, union, difference, image by an affine multidimensional transformation function, supported by PolyLib based on the Minkowski dual representation of a polyhedron,
- integral matrix decomposition algorithms provided by PolyLib and
- parametric integer linear programming provided by PIP or Omega.

For the first item we refer to the on-line documentation of PolyLib [22], [105] while the last two topics are briefly presented in the coming two sections.

2.2 Matrix decompositions

2.2.1 Smith Normal Form (SNF)

It was proved by the English mathematician Henry Smith that any matrix $A \in Z^{m \times n}$ is equivalent to a unique diagonal matrix $S \in Z^{m \times n}$ under unimodular transformations. That is, there exist two matrices $U \in Z^{m \times m}$ and $V \in Z^{n \times n}$ such that:

- $U \cdot A \cdot V = S$,
- $det(U) = -1$ and $det(V) = 1$.
- S is unique and has the form : $\begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$,

where D is a $k \times k$ diagonal matrix with diagonal entries $\delta_i \in Z_+ \setminus \{0\}$.

The matrix S is the *Smith Normal Form (SNF)* of A. The diagonal elements of D are called *the invariant factors* of A.

It has been shown in [52] that the algorithm to compute the SNF of a matrix requires about $O(m^3 n \log \|A\| \; M(m \log \|A\| \;))$ bit operations and $O(nm^2 \log \|A\| \;)$ bit storage, where $\|A\| = max_{ij}|A_{ij}|$, and $M(x)$ represents the bit operations that are required to multiply two x-bit integers; for example using standard arithmetic $M(x) = x^2$.

2.2.2 Hermite Normal Form (HNF)

It was proved by the French mathematician Charles Hermite that for a full row rank integral matrix $A \in Z^{m \times n}$, there always exists a unimodular matrix $C \in Z^{n \times n}$ and a lower triangle integral matrix H such that:

- $A \cdot C = [H, 0]$,
- $\forall i,\ h_{ii} > 0$,
- $\forall (i, j)$ such that $i > j$, $|h_{ij}| \leq h_{ii}$.

The matrix H is the *Hermite Normal Form (HNF)* of A. H has the property that $H^{-1}C$ is an integral matrix. The C matrix in the HNF is not unique. Consider that the number of the zero columns of the $[H, 0]$ matrix is k. Whatever column transformations are applied to the last k columns of C, it will generate another unimodular matrix C' such that $A \cdot C' = [H, 0]$.

According to [52], the algorithm to compute the HNF of a matrix requires about $O(nm^4 \log^2 M)$ bit operations and $O(m^2 \log M)$ bit storage.

2.3 Parametric Integer Linear Programming

The main core technique used in each of our compiler steps is Parametric Integer Linear Programming (PIP). The algorithm is developed by Paul Feautrier and is supported by the PIP/PIPLib library. The algorithm finds in a finite number of steps the lexicographical maximum or minimum among the integers belonging to a convex domain. In this section we briefly introduce the PIP algorithm. The algorithm consists of three steps:

Step1: Use the Lexicographical Dual Simplex algorithm to find the maximum/minimum real lexicographical point over the considered convex domain. If it is infeasible, then so is the integer problem. If the optimal solution is all integer, the integer program is solved as well. Otherwise, go to Step 2.

Step2: Derive a new inequality (or cut) from the current problem constraints which makes the current solution infeasible without eliminating any integer points within the current set of constraints. Add the new constraint to the current set of constraints. The new obtained problem is unfeasible. Go to Step 3.

Step3: If the new linear program is unfeasible then the original problem has no integer solution. If the new optimum is integer, then the integer problem is solved. Otherwise, go to Step 2.

2.3.1 Linear Programming (LP)

In this section we present the Dual Simplex algorithm [25, 81] used in the first step of PIP. This algorithm is used to solve the following linear programming problem:

LP problem : *Minimize the linear function with positive coefficients* $p_i > 0$*:*

$$z(x_1, x_2, ..., x_n) = p_1 x_1 + p_2 x_2 + ... + p_n x_n, \tag{2.2}$$

where $\forall j = 1, ..., n, x_j \geq 0$ *under the constraints given by* m *inequalities:*

$$y_i(x_1, x_2, ..., x_n) = a_{i1} x_1 + a_{i2} x_2 + ... + a_{in} x_n + a_i \geq 0, \tag{2.3}$$

where $i = 1, ..., m$*, and the arbitrary* y_i *represents a slack variable.*

With linear programming terminology, the function z is the *objective* of the problem, whereas the system of inequalities given by the m inequalities y_i is the *subject* of the problem. The problem can be condensed in the following matrix format:

$$\begin{bmatrix} P \\ I_n \\ A \end{bmatrix} x + \begin{bmatrix} 0 \\ a \\ 0 \end{bmatrix} \geq 0, \tag{2.4}$$

where P is a $(1 \times n)$ row matrix with elements p_j, a is a $(m \times 1)$ column matrix made of elements a_i, I_n is the identity matrix and A is a $(m \times n)$ matrix made of the elements a_{ij}. To simplify the presentation further, assume that: $S = \begin{bmatrix} P \\ I_n \\ A \end{bmatrix}$ and $t = \begin{bmatrix} 0 \\ a \\ 0 \end{bmatrix}$.

The algorithm performs a number of Jordan eliminations until a solution is found. Obviously, if $\forall i, a_i \geq 0$ then $x_i = 0, i = 1..n$ is the solution to the problem. However, if this is not the case, then select an index i for which $t_i < 0$ and then an index j for which the vector $S_{\bullet j}/S_{ij}$ is the smallest among those with $S_{ij} > 0$. Those conditions avoid cycling[2] [46, 81]. Next, perform a Jordan elimination over the a_{ij} element. The elimination leads to a new problem described by the table $[S't']$ of whose elements are as follows:

$$\begin{aligned} S'_{\bullet j} &= (1/S_{ij})S_{\bullet j}, \\ S'_{\bullet k} &= S_{\bullet k} - (S_{ik}/S_{ij})S_{\bullet j}, \quad \text{for } k \neq j \\ t' &= t - (t_i/S_{ij})S_{\bullet j}. \end{aligned}$$

It can be seen that both $S'\bullet j$ and $S'\bullet k$ will remain positive. On the other hand $t' > t$ and because there are at most C^n_{m+n} possible eliminations, it turns out that within a fixed number of steps we arrive to a solution. The elimination of variables stops when $t > 0$ and the solution is given by the first n elements of t.

Lexicographic Dual Simplex

The problem of finding the lexicographical minimum point within a set of linear constraints as specified by equation 2.3 can be solved using the above approach. For an arbitrary parametric convex domain $\mathcal{D}$ embedded in an n-dimensional space, there always exists a row vector $C = [c_1, c_2, ..., c_n]$ such that $\forall\ x_i, x_j \in \mathcal{D}, x_i \prec x_j \equiv C\ x_i < C\ x_j$. Thus, C can be used as objective of the classical Dual Simplex problem in order to find the lexicographical maximal point in a convex domain. Observe that the objective row C does not play a role from the point of view of performing the eliminations and, therefore, doesn't have to be specified.

2.3.2 Integer Linear Programming with Gomory cuts

Integer linear programming uses the LP problem given in Section 2.3.1 with the additional integrability added to its solution. Suppose that the optimal solution of the dual simplex problem is not represented by an integral vector. This means $\exists\ i,\ 1 \leq i \leq n$ such that $t_i \notin Z$. Let i be the first index for which

[2] Cycling is the condition that occurs when the simplex method gets "stuck" and finds itself revisiting the same vertices over and over.

$t_i \notin Z$. In this case to arrive to integral solution which maximizes the objective function, the following constraint (i.e., Gomory cut [44]) is added to the subject that excludes the continuous optimum while keeping all feasible integer points:

$$\sum_j \frac{(DS_{ij})\%D}{D} x_j - \frac{(Dt_i)\%D}{D} \geq 0, \tag{2.5}$$

where $\%$ is the remainder operator: $a = b(a\%b) + r,\ 0 \leq r < b$. The new added row has a negative constant term such that to restore the feasibility extra eliminations have to be performed. Based on the observation that the constant term t strictly increases lexicographically at each step, it is proved [46, 81] that in a finite number of steps the algorithm will either find an integral solution or will prove that the problem is unfeasible. Moreover an uniform bound on the number of cuts can be proved [24].

2.3.3 Parametric programming

The previous analysis was done under the assumption that at each step the elements of the column vector b have known values and that their sign can be evaluated. However, in case the constants depend linearly on a number of parameters the analysis can be done in a symbolic way. The index i of the row chosen for elimination is controlled by the rule that t_i is negative. Thus the problem is split up into two subproblems: one for which $t_i \geq 0$ and the other one in which $t_i < 0$. The algorithm continues by building a tree whose leafs will be the parametric solutions. Notice that although the algorithm is guaranteed to terminate in a finite number of steps, the obtained solution is not unique [37].

2.3.4 Empty Domain Test

The ILP technique, consisting of symbolic implementation of the *Lexicographic Dual Simplex* and *Gomory Cuts* algorithms can also be used to determine if a certain domain contains at least one integer point. This represents what we call the *Empty Domain Test* (ET). Basically by determining the lexicographically maximum or minimum of a domain the existence of an integral point in that domain is established. The test makes use of a function that takes as argument a *domain* and returns a boolean: *true* if there are no integer solutions in that domain or otherwise *false* if there is at least one integer solution in that domain:

$$ET(D) = \begin{cases} True, & \nexists x \in D \\ False, & \exists x \in D. \end{cases}$$

As an alternative, the ET problem can be solved using an integer version of the *Fourier-Motzkin elimination* as provided by the Omega library [73], i.e., instead of searching for the lexicographical maximum point, the question whether there exists a single integral point is answered. Note that both algorithms have the same complexity, as the *Fourier-Motzkin elimination* method can be modified to a method for linear programing [74].

Chapter 3

Consumption Restructuring

Restructuring programs for parallel execution involves removal of anti and output dependences. These dependences, so called false dependences, are due to memory reuse. Different methods such as array data flow analysis, variable expansion, and array privatization have addressed the topic of removing false dependences in order to enable vectorization and parallelization [75, 38, 64, 13, 47, 70, 17]. These methods remove all the false dependences by means of deriving equivalent single assignment programs. In single assignment programs each individual use of a certain value requires a separate storage location, thereby significantly increasing the storage requirements. This is for example undesirable in embedded systems. Mapping single assignment programs onto hardware platforms requires sophisticated compile-time [77] and/or run-time memory management support systems [42] to decrease the memory overhead caused by the single assignment format. However, sophisticated run-time memory management systems may lead to performance degradation [42]. This chapter presents a compile time method for removing false dependences, but without requiring single assignment storage formats and thus run-time memory management support.

3.1 Problem Definition

In the Consumption Restructuring step presented below, the program is restructured such that each assignment statement will write each of its output arguments into a dedicated memory array at the same address as in the original code. As a result, all other false dependences present in the original program are removed, except for the output dependences associated with the same argument of the same statement. This transformation is presented in Figure 3.1 using the example program introduced previously in Figure 1.1. First, array $r2$ is replaced by two arrays $r21$ and $r22$. Due to this, extra control that allows to decide whether at a certain execution the assignment statement $F4$ reads data from $r21$ or from $r22$ has to be inserted. Consequently, the iteration space (IS) of statement $F4$ is partitioned into two subdomains. Each subdomain represents what we call an *Input Port Domain (IPD)*. The input port domain IPD1 contains the IPs at which statement $F4$ reads data produced by statement $F2$ and IPD2 contains the IPs at which statement $F4$ reads data produced by statement $F3$. Graphically, the IPDs of an assignment statement are depicted in Figure 3.1 as black spots located at the ends of the incoming edges of the consumer assignment statement. The partitioning of the IS into IPDs is done by adding linear inequalities to the IS of $F4$ as shown in Figure 3.1. The next section presents an ILP procedure for automatic generation of such a partitioning.

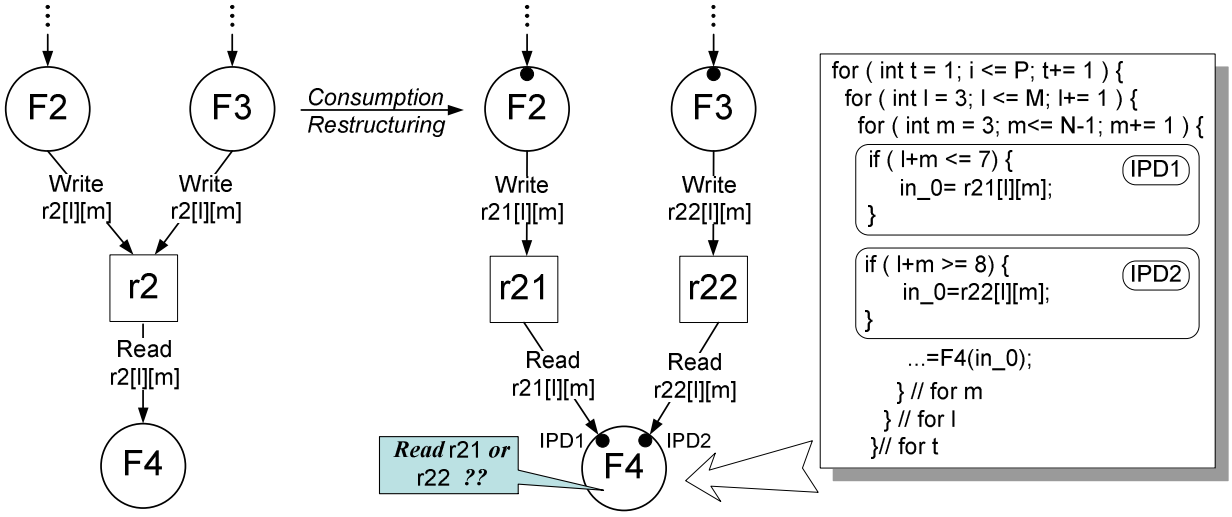


Figure 3.1: Consumption restructuring - need for dependences analysis

3.2 Solution - ILP approach

To restructure the IS of an arbitrary consumer statement C_j with respect to a read from a memory array r, we first identify the sets of producer statements that write to r. Let S_r be the set of all assignment statements P_i that write data to the memory array r from where statement C_j may consume it. For each statement P_i, we replace the write to the array r with a write to a separate array r_i. To maintain correct program execution, the statement C_j has to consume (read) data from the new memory array r_i. Therefore, we have to connect the consumption of an arbitrary data token to its production. To solve this problem we propose a four step approach, of which the first three steps represent array data flow analysis [38], based on which, in the fourth step, the program restructuring takes place.

step 1: In the first step we compute the flow dependency from each producer $P_i \subset S_r$ to the consumer C_j due to an input argument in, independently of other writes. Suppose that the iteration domain of statement P_i is given by the set $\mathcal{P}_i$ and that at iteration $x_p \in \mathcal{P}_i$ data is written in the array r at location $write(x_p)$. Similarly the iteration domain of the consumer statement C_j is given by the set $\mathcal{C}_j$ and at the iteration point $y_c \in \mathcal{C}_j$ data is read from array r from the location $read(y_c)$:

$$\begin{array}{lcl} P_i : ..., \ r(write(x_p)), \ ... & = & \ldots \\ & \vdots & \\ C_j : \quad ... & = & ..., in = r(read(y_c)), ... \end{array}$$

Using the total order operator " $\prec_t$" (introduced in Chapter 2), we now represent the subset of $\mathcal{C}_j$ made of iterations at which data consumed as argument in was produced by P_i:

$$S^{in}_{P_i} = \{y_c \in \mathcal{C}_j \mid \exists \ x_p \ s.t. \ x_p \in \mathcal{P}_i \wedge x_p \prec_t y_c \wedge write(x_p) = read(y_c)\}. \tag{3.1}$$

Our target is to express the set $S^{in}_{P_i}$ in terms of the consumer iterators. To do this, we have to eliminate x_p from the definition of $S^{in}_{P_i}$. The elimination is achieved by solving the following PIP problem:

$$\begin{array}{rll} \textbf{subject to:} & y_c \in \mathcal{C}_j, & (c1) \\ & x_p \in \mathcal{P}_i, & (c2) \\ & x_p \prec_t y_c, & (c3) \\ & write(x_p) = read(y_c), & (c4) \\ \textbf{objective:} & x_m(y_c) = max_{\textbf{lex}}\{x_p\}. & \end{array} \tag{3.2}$$

Although the total order operator " $\prec_t$" does not represent a linear constraint, the presented problem can be solved by integer linear programming techniques [38]. Observe that the conditions $(c1), (c2), (c3)$ and $(c4)$ represent the constraints that define the set $S^{in}_{P_i}$. Because the PIP problem is set to compute the lexicographically maximum, the solution of the PIP problem i.e., $x_m(y_c) = max_{\textbf{lex}}\{x_p\}$ represents for an arbitrary point y_c the last write that took place in array r at address $read(y_c)$. The solution of the PIP problem is expressed as a multistage conditional expression:

$$\begin{array}{rl} \text{if } (y_c \in \mathcal{D}_1), & \\ & \text{then} \quad x_m = T_1(y_c), \\ \text{else if } (y_c \in \mathcal{D}_2), & \\ & \text{then} \quad x_m = T_2(y_c), \\ & \vdots \\ \text{else if } (y_c \in \mathcal{D}_n), & \\ & \text{then} \quad x_m = T_n(y_c), \end{array}$$

where $\mathcal{D}_1, ..., \mathcal{D}_n$ are disjoint parameterized polytopes and $T_1, ..., T_n$ are affine transformations. There can be branches $\mathcal{D}_k$ for which the solution is not defined i.e., $T_k = \perp$. Such a case means that if $y_c \in \mathcal{D}_k$ then there does not exist x_p such that $x_p \in \mathcal{P}_i \wedge write(x_p) = read(y_c)$. In other words, for points belonging to $\mathcal{D}_k$ the source of data doesn't come from statement P_i. Hence, the dependency which exists between the two statements without taking into account the other possible statements that write to the same memory array is expressed as follows:

$$S^{in}_{P_i} = \bigcup_{i \in I} \mathcal{D}_i, \ where\ I = \{i \mid T_i \neq \perp\}. \tag{3.3}$$

step 2: For each consumer, combine the direct dependences obtained in step 1 from multiple writes. The result is a solution tree that combines the solution trees coming from different producer assignments. For more information about how to combine different solution trees we refer to [38].
step 3: Remove all the unfeasible leafs (domains) from the obtained solution tree. Finding the unfeasible leafs in a tree is equivalent to deciding whether a convex set contains integer points. Again this problem can be solved using PIP. Finally, the branches of this tree (of which solutions point to one and the same producer statement) represent a set of which integral restriction define what we call an input port domain (IPD). Formally an IPD is expressed as follows:

Definition 1 Input Port Domain (IPD):

$$\begin{aligned} IPD = \{y_c \in \mathcal{C}_j \mid \exists\ x_p\ s.t.\ x_p \in \mathcal{P}_i\ \wedge\ x_p \prec_t y_c\ \wedge\ write_{Pi}(x_p) = read_{Cj}(y_c)\ \wedge \\ \forall\ k \neq j,\ \neg\ \exists\ x'_p \in \mathcal{P}_k\ \wedge x_p \prec_t x'_p \prec_t y_c\ \wedge\ write_{Pk}(x'_p) = read_{Cj}(y_c)\}, \end{aligned} \tag{3.4}$$

where $\mathcal{P}_i$, $\mathcal{P}_k$, and $\mathcal{C}_j$ represent the iteration spaces of the statements P_i, P_k, and C_j, respectively, and $write_{Pi}$ and $read_{Cj}$ represent the write access for producer statements P_i and the read access of the consumer statement C_j, respectively.

step 4: Repeat steps $1-3$ to compute the IPDs of all the program assignment statements. Then, use a dedicated memory array to communicate data between each data dependent pair of statements. This is done by replacing the original program memory allocation with one where each dependency defined over an IPD exists over a dedicated memory array, without changing the read and write addressing functions. For the correctness of this transformation we refer to [38, 36].

By performing the four steps for each producer/consumer pair of statements, we restructure the consumption of data over the whole program. Due to the lexicographical maximum objective of the PIP problem stated by 3.2, for each data consumption we establish exactly the corresponding producer iteration x_m. Depending on a certain IPD to which y_c belongs, x_m is obtained by computing an affine dependency function T_i. The iteration domains corresponding to a unique source of data represent an IPD. Thus, *each IPD represents a union of parameterized polytopes where each polytope includes all the IPs at which an input argument of an assignment statement is being produced by one and the same statement*. Without losing generality, from now on we assume that each IPD is represented by only one parameterized polytope $\mathcal{C}(N)$, such that: $IPD = \mathcal{C}(N) \cap Z$. Thus, with respect to each data argument, a P/C pair of statements is uniquely defined by a polytope $\mathcal{C}(N)$ together with an affine dependency function f represented by an integral matrix M, and an offset vector O, i.e., $f(x) = Mx + O$.

Running example:

Consider the code given in Figure 1.1, where statements *stm2* and *stm3* are responsible for writing data to a 2-d array $r2$ from where statement *stm4* consumes data. In the original application, we identify the P/C pairs of assignment statements $PC_1 = (F2, F4)$ and $PC_2 = (F3, F4)$, each of them communicating data via the global array $r2$. We replace in each P/C pair the write to array $r2$ at location $r2[l][m]$, with a write to array $r21$ at location $r21[l][m]$, and to array $r22$ at location $r22[l][m]$, respectively. As a consequence, assignment statements $F2$ and $F3$ will write data to separate storage arrays as shown in the right-hand side of Figure 3.1.

To keep the execution of the network correct, we have to find at each execution of assignment statement $F4$ the location in $r21$ or $r22$ containing the appropriate input data. This correspondence is obtained using the data-dependency functions corresponding to the P/C pairs of statements $(stm2, stm4)$ and $(stm3, stm4)$. According to step 1 of our approach, since we have two P/C pairs, the following two PIP problems (corresponding to PC_1 and to PC_2) have to be solved:

Problem PC1:

subject to:
$1 \le t_c \le P$, (c1) $\quad 1 \le t_p \le P$, (c2)
$3 \le l_c \le M$, $\quad 3 \le l_p \le M$,
$3 \le m_c \le N-1$, $\quad 3 \le m_p \le N-1$,
$l_p + m_p \le 7$,
$(t_p, i_p, j_p) \prec_t (t_c, l_c, m_c)$ (c3)
$(l_p, m_p) = (l_c, m_c)$ (c4)

objective: $\max_{lex}\{t_p, l_p, m_p\}$

Problem PC2:

subject to:
$1 \le t_c \le P$, (c1) $\quad 1 \le t_p \le P$, (c2)
$3 \le l_c \le M$, $\quad 3 \le l_p \le M$,
$3 \le m_c \le N-1$, $\quad 3 \le m_p \le N-1$,
$l_p + m_p \ge 8$,
$(t_p, i_p, j_p) \prec_t (t_c, l_c, m_c)$ (c3)
$(l_p, m_p) = (l_c, m_c)$ (c4)

objective: $\max_{lex}\{t_p, l_p, m_p\}$

Although the relation $(c3)$ from from 3.2 does not represent a linear constraint, based on relation 2.1 from Chapter 2, it can be decomposed as follows: $(t_p, l_p, m_p) \prec_t (t_c, l_c, m_c) \equiv ((t_p, l_p, m_p) \prec (t_c, l_c, m_c)) \vee ((t_p, l_p, m_p) = (t_c, l_c, m_c) \wedge T(x_p, y_c) = 1)$. On the other hand, looking at the code depicted in Figure 1.1 one can easily see that if $(t_p, l_p, m_p) = (t_c, l_c, m_c)$ then $T(x_p, y_c) = 1$. Hence, $(t_p, l_p, m_p) \prec_t (t_c, l_c, m_c) \equiv ((t_p, l_p, m_p) \prec (t_c, l_c, m_c)) \vee ((t_p, l_p, m_p) = (t_c, l_c, m_c))$. Further explanations about how the lexicographical operator " $\prec$" is decomposed into a series of linear constraints can be found in Section 5.2.3 from Chapter 5. As a result of this decomposition, the two PIP problems are decomposed into a number of ILP problems for which a number of solution trees exist. Following a similar approach as the one used at step 2, those individual solution trees are combined into the following final solution trees ST_1 and ST_2:

Solution Tree ST1:

```
if (1≤ t_c ≤P) {
  if (3≤ l_c ≤M) {
    if (3≤ m_c ≤N-1) {
      if (l_c + m_c ≤7) {
Sol : (t_p, l_p, m_p) = (t_c, l_c, m_c);
}}}}
```

Solution Tree ST2 :

```
if (1≤ t_c ≤P) {
  if (3≤ l_c ≤M) {
    if (3≤ m_c ≤N-1) {
      if (l_c + m_c ≥8) {
Sol : (t_p, l_p, m_p) = (t_c, l_c, m_c);
}}}}
```

The domains where the two dependency functions (specified as **Sol**) are valid, represent disjoint sets (defined by the **Sol** enclosing if statements), as the constraint $l + m \leq 7$ contradicts the constraint $8 \leq l + m$. Further on, following steps 2 and 3 we get that each of the two domains represents an input port domain. Hence, at an IP (t, l, m) belonging to $IPD_1 = \{(t, l, m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l + m \leq 7\}$, the statement $F4$ consumes data stored in $r21[l][m]$ which was produced by $F2$ at IP (t, l, m). Thus, the PC_1 dependency function is $f_{PC_1}(t, l, m) = (t, l, m)$. Similarly, at an IP (t, l, m) belonging to $IPD_2 = \{(t, l, m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 8 \leq l + m\}$, the statement $F4$ consumes data stored in $r22[l][m]$ which was produced by $F3$ at IP (t, l, m). Hence, $f_{PC_2}(t, l, m) = (t, l, m)$.

At this step we derive also the dependency functions corresponding to the PC pairs $PC_3 = (F1, F2)$ and $PC_4 = (F1, F3)$. They are $f_{PC_3}(t, l, m) = (t, l-2, m+1)$ valid on $IPD_1 = \{(t, l, m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l + m \leq 7\}$ and $f_{PC_4}(t, l, m) = (t, l-1, N)$ valid on $IPD_1 = \{(t, l, m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 8 \leq l + m\}$. The dependency functions will be used in future steps of our compilation process. □

3.3 An alternative approach

Another possibility to compute the IPDs, and therefore, to realize the Consumption restructuring step, is to make use the Omega test and Presburger formulas [73]. The Omega test is an integer linear programming modification of the Fourier-Motzkin elimination. As presented in [75], to formulate the data dependence problem, a technique called *exact value-based array data dependence analysis* is used. The IPD definition given in (3.4) can be computed using the Omega test and Presburger formulas. To arrive at a formulation of the IPD depending only on y_c, a number of eliminations have to be performed. Those eliminations are done using the Omega test repetitively. To avoid an eventual explosion of tests that have to be calculated a number of techniques [73, 75] have been developed.

Based on the Omega test, it is possible also to get the precise source of data. The idea is that the Omega test can be used to find the lexicographically maximal point within a parametric domain

$\mathcal{C}(N)$, as given by the next Presburger formula:

$$max_{lex}\{x \in \mathcal{C}(N)\} \equiv \{x \in \mathcal{C}(N) \wedge (\ \neg\ \exists\ i\ s.t.\ x \prec i \vee i \in \mathcal{C}(N))\}. \tag{3.5}$$

The left-hand side of relation 3.5 is the lexicographical maximal point, while the right-hand side represents the set containing as integer points only the lexicographical maximal point. The Omega test is decisive with respect to the existence of integer solutions to affine constraints by finding an integer that respects the constraints. Thus, when applying the Omega test on the set defined by the right-hand side of relation 3.5, we obtain the lexicographical maximal point. However, this feature is not directly supported by the Omega library and, therefore, for our compiler development we have chosen PipLib as the main ILP tool.

Example - lexicographically maximum using the Omega test and Presburger formulas

Suppose we want to find the lexicographically maximal point inside the following domain constrained by the integer parameters N and M.

$$x_m = max_{lex}\{x \in Z \mid 0 \leq x \wedge x \leq N \wedge x \leq M\} \equiv \{x \in Z \mid 0 \leq x \wedge x \leq N \wedge x \leq M \wedge \\ \neg\ \exists(\ i\ s.t.\ x \prec i \vee 0 \leq i \wedge i \leq N \wedge i \leq M)\}.$$

Here, because of the one dimensional domain we have that: $x \prec i \equiv x < i$. Thus, by eliminating i we get that:

$$\begin{aligned} x_m &= \{x \in Z \mid 0 \leq x \wedge x \leq N \wedge x \leq M \wedge \neg(x < N \wedge x < M)\} \\ &= \{x \in Z \mid 0 \leq x \wedge x \leq N \wedge x \leq M \wedge (x \geq N \vee x \geq M)\} \\ &= \{x \in Z \mid x = N \wedge 0 \leq N \leq M\} \vee \{x \in Z \mid x = M \wedge 0 \leq M \leq N\}. \end{aligned}$$

This result is interpreted as follows:

$$x_m = \begin{cases} N\ if\ 0 \leq N \leq M, \\ M\ if\ 0 \leq M \leq N. \end{cases}$$

3.4 Discussion

In this chapter we have presented a parallelization technique which removes all the program anti dependences as well as all the output dependences originating from different assignment statements. Notice that the method does not remove the output dependences originating from the same assignment statement. The technique allows us to restructure the data consumption in a nested loop program, and can be implemented using two classical integer linear programming tests i.e, PIP or Omega. These tests are used to express the domain of existence of a dependency between two program statements. Although the dependency function is not needed in this step, it can be calculated by setting the objective of the ILP problem in the presented techniques to be the lexicographic maximum. In general, integer linear programming is expensive in terms of memory and computation time. However, there are cases in which cheaper techniques can be involved. For example, for loops with certain array indexing such as Zero Index Variable (ZIV) subscripts and single index variable (SIV) subscripts the independence between two statements can be proved based on less expensive techniques like GCD or Banerjee test. More complex and accurate tests, e.g., Gamma or Delta, are applied for Multiple Index Variable (MIV) array subscripts. A cascade of tests combining the previous tests progressively increasing in accuracy as well as in time and memory requirements can be found in the literature [65, 43, 63]. As a future work we plan to implement such a cascade test in our compiler including PIP/Omega as a back-up test for the exact dependency.

Chapter 4

Production Restructuring

After performing the Consumption Restructuring step, two assignment statements from the original program that were storing data into the same memory array, now store it into separate arrays. As a consequence, from the writing (production) point of view they become independent of each other. On the other hand, although two assignments that read (consume) data stored at the same memory address, can be executed independently of each other (input dependences do not restrict program execution ordering), there are two parallelization aspects involved here:

- To allow parallelism, the target architecture has to provide support for multi-port memories. In general those memories are expensive and have a limited number of ports that are accessible in parallel.
- By preserving the output dependences with respect to a producer argument, although independent of each other, the corresponding consumer statements are enforced to respect a certain interdependency. Consider the producer statement $F1$ with two consumers $F2$ and $F3$ as shown at the left side of Figure 4.1. $F1$ can not overwrite a memory location $r1[i][j]$ before both $F2$ and $F3$ have finished to consume from that location. Implicitly, a future consumption performed by $F2$ from location $r1[i][j]$, can not take place before $F3$ has finished to consume the previous data. Hence, $F2$ and $F3$ executed at different IPs are interdependent.

To address the previous two aspects, in this chapter we investigate how to remove some of the program input dependences and thereby expose more parallelism without being constrained by multiple-port memory support in the target architecture.

4.1 Problem Definition

During Production Restructuring, the program is restructured so that each assignment statement will read each of its input arguments from a dedicated memory array, from the same address as in the original code. As a result we remove all input dependences remaining in the code after running Consumption Restructuring, except for the input dependences related to the same argument of the same statement. The transformation is illustrated in Figure 4.1, where array $r1$ is replaced by two different arrays $r11$ and $r12$. Due to the restructuring, statement $F1$ now has to decide at each execution whether to write data to $r11$, to $r12$, to both arrays, or to none of the arrays.

The restructuring of the program partitions the IS of each producer statement into a number of subdomains which may overlap. Each subdomain represents what we call an *Output Port Domain (OPD)*. In our example, OPD1 contains the IPs at which statement $F1$ produces data consumed

by statement $F2$, and OPD2 contains the IPs at which statement $F1$ produces data consumed by statement $F3$. Graphically, the OPDs are represented in Figure 4.1 as black spots located at the beginning of the outgoing edges of each producer statement. The partitioning of the IS into OPDs is done by adding linear inequalities to the IS of $F1$, as shown in the code at the right of Figure 4.1. In the following sections we present a procedure for partitioning of a producer IS into OPDs based on integer linear programming and matrix decomposition algorithms.

4.1.1 Formal representation

To derive the inequalities that partition the producer domain into OPDs, we first identify the sets of consumer input arguments that are read from the same memory array. Let D_r be the set made of the input arguments arg_i^r that are read from array r. For each argument arg_i^r, we replace the read of data from the global array r with a read from a separate array r_i. Due to the Consumption Restructuring step, there is only one assignment statement P^r that writes data into r, i.e., all the output dependences with respect to two different assignment statements have been removed. Thus, to have a correct execution, the producer statement P^r has to decide at each IP y what are the proper arrays r_i where data has to be stored. Consider now a generic P/C pair of statements P^r/C^r, where at least one of the C^r input arguments is read from the array r. This P/C pair is characterized by an affine data-dependency function f and by a parametric consumer $IPD = \mathcal{C}(N) \cap Z^n$, as obtained after the Consumption Restructuring step. Finding the appropriate storage array for an arbitrary producer IP y, is equivalent with deciding whether y belongs to the following set formally referred as output port domain:

Definition 2 Output Port Domain (OPD):

$$OPD = f(\mathcal{C}(N) \cap Z^n) = \{y \mid \exists\, x \in \mathcal{C}(N) \cap Z^n \;\; s.t.\; y = f(x)\}. \tag{4.1}$$

According to this definition, OPD is a *linearly bounded lattice (LBL)* [89], i.e., it represents the integral image of $\mathcal{C}(N)$ through the affine function f[1]. Without loss of generality, we assume that f is a linear function represented by an $(m \times n)$ integral matrix $M \in Z^{m \times n}$, where m represents the dimension of the producer IS.

Relation (4.1) implies that in order to decide whether an arbitrary integral point y belongs to OPD the following objects have to be taken into account:

- the dependency function f and
- the Consumer Input Port Domain $\mathcal{C}(N) \cap Z^n$.

In Section 4.2 we present a general solution to this problem that takes into account both the dependency function and the Consumer domain. However, in Section 4.3 we will show that under certain circumstances the integrality of the Consumer domain does not have to be taken into consideration. This observation leads to a less computationally complex procedure based on Smith Normal Form. Finally, in Section 4.4 we merge the two above approaches into a hybrid approach.

[1]Notice that $f(\mathcal{C}(N) \cap Z^n)$ is different than the set made of the integral points included in the real image of $\mathcal{C}(N)$ through the function f, i.e.,$f(\mathcal{C}(N) \cap Z^n) \neq f(\mathcal{C}(N)) \cap Z^m$ (see Example 2 from page 37).

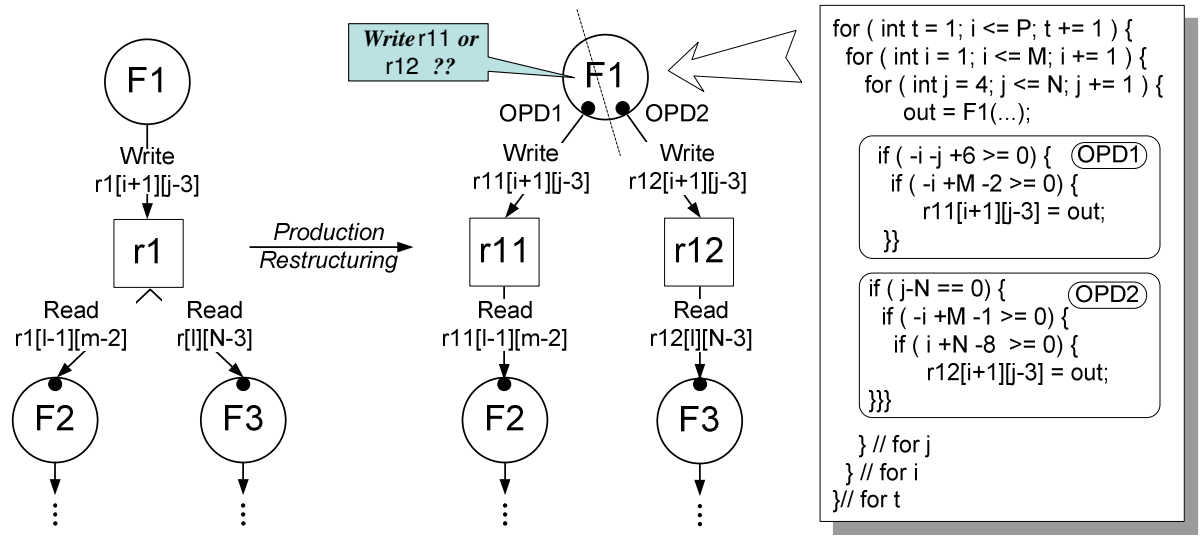


Figure 4.1: Production Restructuring

4.2 Solution - ILP approach

To determine whether an arbitrary point y belongs to the set described in 4.1, we make use of the following theorem:

Theorem 2 *Let the following parametric integer linear programming (PIP) problem [37] be given, with variable x and parameter y[2]:*

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{C}(N), && (c1) \\ & y = f(x), && (c2) \\ \textbf{objective:} \quad & x_m(y) = min_{\textbf{lex}}\{x\}, \end{aligned}$$

where $\mathcal{C}(N)$ is a parametric polytope and f is an affine integral function. Let the solution of the PIP problem be expressed by the following multistage conditional expression:

$$\begin{aligned} \textit{if } (y \in \ \mathcal{D}_1), & \\ & \textit{then} \quad x_m = \ T_1(y), \\ \textit{else if } (y \in \ \mathcal{D}_2), & \\ & \textit{then} \quad x_m = \ T_2(y), \\ & \vdots \\ \textit{else if } (y \in \ \mathcal{D}_n), & \\ & \textit{then} \quad x_m = \ T_n(y), \end{aligned}$$

where $\mathcal{D}_1, ..., \mathcal{D}_n$ are disjoint parameterized polytopes[3] and $T_1, ..., T_n$ are affine transformations. Then, $OPD = \cup_{i \in I} \mathcal{D}_i$, where $I = \{i \mid T_i \neq \perp\}$.

[2] Observe that the objective function depends on y.

[3] Although the domains used by PIP to express the solution tree are LBLs by nature, we can represent them as polytopes by introducing extra dimensions for each of the div statements used as filters.

Proof: The proof is straightforward: $y \in OPD \Leftrightarrow C(y) \neq \emptyset$, where $C(y) = \{x \in \mathcal{C}(N) \cap Z^n \mid \text{s.t. } y = f(x)\}$. However, $C(y) \neq \emptyset \Leftrightarrow \exists x_m = \min_{lex} C(y) \Leftrightarrow \exists k$ such that $y \in \mathcal{D}_k$ and $x_m = T_k(y)$. □

Observation 1: The solution branches of the PIP problem stated in Theorem 2 can arrive in an empty statement, i.e., $T_i = \perp$. This situation corresponds to the case when f is not surjective which means that there are producer IPs at which the produced data is not consumed.
Observation 2: Although we are interested only in the existence of an integral solution, we choose as objective the lexico-minimal or maximal function. This allows us to gather additional information further used during the Process Network Synthesis step to optimize the network memory management.

Running example

Let us analyze how the Production Restructuring step applies to the running example depicted in Figure 1.1. As shown in Figure 4.1, we have two P/C pairs that are sharing the same memory array: $PC_3 = (F1, F2)$ and $PC_4 = (F1, F3)$. As a consequence, we have to partition the IS of $F1$ into two output ports: $OPD1$ consisting of the IPs at which data has to be loaded into $r11$ and $OPD2$ consisting of the IPs at which data has to be loaded into $r12$. To do the partitioning we make use of the information gathered during Consumption Restructuring (see the Running example from page 21). Namely we use the PC_3 and PC_4 dependences functions: $f_{PC_3}(t,l,m) = (t, l-2, m+1)$ defined on $IPD_1 = \{(t,l,m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l+m \leq 7\}$ and $f_{PC_4}(t,l,m) = (t, l-1, N)$ defined on $IPD_1 = \{(t,l,m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 8 \leq l+m\}$. Thus, the following two PIP problems (corresponding to PC_3 and PC_4) have to be solved:

Problem PC3:

subject to:
$$1 \leq t_c \leq P, \quad 3 \leq m_c \leq N-1, \quad (c1)$$
$$3 \leq l_c \leq M, \; l_c + m_c \leq 7,$$
$$(t_p, i_p, j_p) = (t_c, l_c - 2, m_c + 1), \quad (c2)$$

objective: $\min_{lex}\{t_c, l_c, m_c\}$.

Problem PC4:

subject to:
$$1 \leq t_c \leq P, \quad 3 \leq m_c \leq N-1, \quad (c1)$$
$$3 \leq l_c \leq M, \; 8 \leq l_c + m_c,$$
$$(t_p, i_p, j_p) = (t_c, l_c - 1, N), \quad (c2)$$

objective: $\min_{lex}\{t_c, l_c, m_c\}$.

The two ILP problems can be solved using algorithms like Lexicographical Dual Simplex and Gomory Cuts or an integer version of Fourier-Motzkin Elimination [37, 75]. By solving them we get the solution trees ST_1 and ST_2, respectively:

Solution Tree ST1:

```
if (1 ≤ t_p ≤ P) {
  if (1 ≤ i_p ≤ M - 2) {
    if (4 ≤ j_p ≤ N) {
      if (i_p + j_p ≤ 6) {
Sol :   (t_c, l_c, m_c) = (t_p, i_p + 2, j_p - 1);
      }
    }
  }
}
```

Solution Tree ST2 :

```
if (1 ≤ t_p ≤ P) {
  if (1 ≤ i_p ≤ M - 1) {
    if (j_p - N == 0) {
      if (i_p + N - 8 ≥ 0) {
        if (2 ≤ i_p ≤ 4) {
Sol1 :    (t_c, l_c, m_c) = (t_p, i_p + 1, - i_p + 7);}
        if (5 ≤ i_p) {
Sol2 :    (t_c, l_c, m_c) = (t_p, i_p + 1, 3);}
}}}}
```

ST_1 and ST_2 are used to partition the IS of statement $F1$ with respect to data production. While initially the data produced by $F1$ was always written at $r[i+1][j-3]$, now depending on the constraints specified by the branches of $ST1$ and $ST2$, the data is written at $r11[i+1][j-3]$ and/or into $r12[i+1][j-3]$. $ST1$ has only one solution defined on the output port domain: $OPD_1 = \{(t_p, i_p, j_p) \in Z^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-2, 4 \leq j_p \leq N, i_p + j_p \leq 6\}$ However, $ST2$ has two solutions, corresponding to two disjoint domains: $\mathcal{D}_1 = \{(t_p, i_p, j_p) \in Z^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8, 2 \leq i_p \leq 4\}$ and $\mathcal{D}_2 = \{(t_p, i_p, j_p) \in Z^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8, 5 \leq i_p\}$. We interpret this as follows: if $ip_1 = (t_p, i_p, j_p)$ is a producer IP belonging to $\mathcal{D}_1$ at which F_1 produces the token t, then $(t_p, i_p+1, -i_p+7)$ represents the first (lexicographically smallest) consumer IP that consumes the token t. Similarly, if ip_2 is a producer IP belonging to $\mathcal{D}_2$ then $(t_p, i_p+1, 3)$ is the first IP that consumes it. However, for the restructuring of a producer IS the information regarding different first consumptions is irrelevant i.e., it matters only whether with respect to a consumer process, a produced token has to be sent or not. Therefore, as shown by Figure 4.1, two disjoint domains $\mathcal{D}_1$ and $\mathcal{D}_2$ are collapsed into a single domain that represents the output port domain $OPD_2 = \mathcal{D}_1 \cup \mathcal{D}_2 = \{(t_p, i_p, j_p) \in Z^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8\}$.

4.3 An alternative solution - the SNF approach

As presented in the previous section, the Production Restructuring step can always be accomplished using parametric integer linear programming. As an alternative to the solution given in the previous section, we now present a less computational intensive procedure that uses the Smith Normal Form (SNF) decomposition. Details about SNF are given in Section 2.2.1. This alternative procedure is based on Theorem 3 from below, and can be applied when the P/C dependency function f is represented by a full column rank matrix [4] M. Without loss of generality we assume that f is linear, i.e., no offset will be taken into account. To introduce Theorem 3, we first recall that according to SNF, a matrix M can be decomposed as follows: $M = U^{-1}\ S\ V^{-1}$, where $S = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$.

Theorem 3 *Let the dependency between Consumer and Producer be represented by an integral full column rank matrix $M \in Z^{m \times n}$. Let*

$$P = \{y \mid y \in M(\mathcal{C}(N)) \ \wedge \ D^{-1}\ U_1\ y \in Z^n\}, \tag{4.2}$$

where U_1 is a matrix made of the first n rows of the matrix U from the SNF of M. Then, $P = OPD$.

Proof: The proof consists of two parts: first we show that a) $OPD \subset P$ and then that b) $P \subset OPD$.

a) Let $y \in OPD$. According to definition (4.1), $\exists x \in \mathcal{C}(N) \cap Z^n$ s.t. $y = M\ x$. Hence, $y \in M(\mathcal{C}(N) \cap Z^n) \Rightarrow y \in M(\mathcal{C}(N))(1)$. On the other hand, using the SNF of the mapping matrix $M \Rightarrow y = U^{-1}\ S\ V^{-1}\ x$. Since M is full column rank $\Rightarrow S = \begin{bmatrix} D \\ 0 \end{bmatrix}$. Thus, $V\ D^{-1}\ U_1\ y = x$, where U_1 is the submatrix consisting of the first n rows of matrix U, where $U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix}$. Because $x \in Z^n \Rightarrow$

[4] An $(m \times n)$ matrix M has full column rank iff M has rank n. Similarly, M has full row rank iff M has rank m.

$V\ D^{-1}\ U_1\ y \in Z^n$. However, V is integral and unimodular, and therefore, $D^{-1}\ U_1\ y \in Z^n$ (2). Hence, from (1) and (2) $\Rightarrow OPD \subset P$.

b) Let $y \in P$. Then $D^{-1}\ U_1\ y \in Z^n$ and therefore, $\exists\ x \in Z^n$ (3), $x = V\ D^{-1}\ U_1\ y$(4). However, y belongs to the real image of the polytope $\mathcal{C}(N)$ through the mapping M, i.e., $y \in M(\mathcal{C}(N))$, and therefore, $U_2\ y = 0_{m-n}$(5). From (4) and (5) it follows that x is such that $y = M\ x$ (6). According to Definition 2 we still have to proof that $x \in \mathcal{C}(N)$.

We observe that $M = U^{-1} \begin{bmatrix} D \\ 0 \end{bmatrix} V^{-1}$ and therefore, $V\ D^{-1}\ U_1\ M = I_n$ (7). But $y \in M(\mathcal{C}(N))$ and therefore using (4) we obtain that $x \in V\ D^{-1}\ U_1\ M(\mathcal{C}(N)) \Rightarrow x \in \mathcal{C}(N)$ (9). Hence, from (3), (6) and (9) it follows that $P \subset OPD$. □

Based on Theorem 3 we find a simple way to derive an OPD. Basically we need to obtain the real image [5] of the polytope $\mathcal{C}(N)$ through the dependency function f and then to filter some of the producer points by verifying (at compile time) whether $D^{-1}\ U_1\ y \in Z^n$.

Example

Here we present an example where the approach based on SNF matrix decomposition can be successfully applied. We have to derive the *if-statements* representing the producer OPD. Consider the code given in Figure 4.2.

Producer:

```
for (int ip=1; ip<=3*N+2*K; ip++){
 for (int jp=1; jp<=4*K+2; jp++){
  for (int kp=1; kp<=K; kp++){
   r[ip,jp,kp]=Init(...);
  }
 }
}
```

Consumer:

```
for (int ic=1; ic<=N; ic++){
 for (int jc=1; jc<=K; jc++){
  Sink[ic,jc]=Fc( r[3*ic+2*jc,4*jc,jc] );
 }
}
```

Figure 4.2: A P/C program before the Production Restructuring step

In this case, we have the Consumer described by the integer points inside the following parameterized polytope:

$$\mathcal{C}(N,K) = \{(i_C, j_C) \in R^2 \mid 1 \leq i_C \leq N \wedge 1 \leq j_C \leq K\}.$$

The data dependency function between consumption and production is represented by the following mapping matrix:

$$M = \begin{bmatrix} 3 & 2 \\ 0 & 4 \\ 0 & 1 \end{bmatrix}.$$

[5] Currently for performing the real image of a polytope we make use of the procedure which is provided by PolyLib and is based on Chernikova's algorithm. Additionally, less computational intensive procedures can be used as well (e.g.,), which in turn may provide less compact domain image representations.

The real domain image of the Consumer domain through the mapping described by matrix M, is represented as follows:

$$\mathcal{D}(N,K) = \{(i_P, j_P, k_P) \in R^3 \| \ 1 \le i_P \le 3*N + 2*K \ \wedge \ 1 \le j_P \le 4*K + 2 \ \wedge \\ 1 \le k_P \le K \ \wedge j_P - 4*k_P = 0 \ \wedge i_P - 2*k_P - 3 \ge 0 \\ \wedge -i_P + 2*k_P + 3*N \ \ge 0\}. \qquad (4.2')$$

However, this domain does not represent the correct OPD as extra IPs have to be filtered out. This happens because the dependency function is not surjective. To derive the producer filter, we make use of Theorem 3. Using the SNF of M we get the following matrices:

$$U^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -4 & 4 & 1 \\ -1 & 1 & 0 \end{bmatrix}, \quad S = \begin{bmatrix} D \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ 0 & 0 \end{bmatrix}, \quad V^{-1} = \begin{bmatrix} 3 & 2 \\ 1 & 1 \end{bmatrix},$$

such that by inverting U^{-1} and D:

$$D^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1/3 \end{bmatrix}, \quad U_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Thus, $D^{-1}\ U_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 0 & 1/3 \end{bmatrix}$, and according to the integrity condition (4.2) ($D^{-1}\ U_1\ y_P \in Z^n$), we have:

$$i_P \in Z, \qquad (4.3)$$

$$1/3\ i_P + 1/3\ k_P \in Z. \qquad (4.4)$$

By default $y_P = (i_P, j_P, k_P)$ is an integral point as it belongs to the IS described by nested for-loops. To derive the Producer domain, we have to add the condition $1/3\ i_P + 1/3\ k_P \in Z$ to those given by the system of inequalities (4.2'). As a result the P/C pair is restructured as shown in Figure 4.3, where the condition $1/3\ i_P + 1/3\ k_P \in Z$ is actually expressed by the semantically equivalent if-statement $(ip + kp)\%3 == 0$.

Producer:

```
for (int ip=1; ip<=3*N+2*K; ip++){
 for (int jp=1; jp<=4*K+2; jp++){
  for (int kp=1; kp<=K; kp++){
   if(jp-4*kp==0){
    if(ip-2*kp-3>=0){
     if(-ip+2*kp+3*N>=0){
      if((ip+kp)%3==0){              //see relation (4.4)
       r[ip,jp,kp]=Init(...);
    }}}}
}}}
```

Consumer:

```
for (int ic=1; ic<=N; ic++){
 for (int jc=1; jc<=K; jc++){
  Sink[ic,jc]=Fc( r[3*ic+2*jc,4*jc,jc] );
 }
}
```

Figure 4.3: The refined P/C program, after applying SNF Production Restructuring approach

4.4 Putting things together - a hybrid approach

In the Production Restructuring step, when partitioning the producer IS, we would like to make use of the Smith Normal Form approach as much as possible for the following two reasons:

- The solution tree obtained from the SNF approach is more compact than the one given by the ILP formulation, i.e., contains less if-statements.
- According to Section 2.2.1, the SNF algorithm is of polynomial complexity while ILP is not.

As presented in Figure 4.4, we arrive at a simple Production Restructuring approach based on a decision tree that combines the two approaches presented so far. We first check whether the dependency matrix is of full column rank. If this is the case we use the SNF approach; otherwise we use the ILP approach.

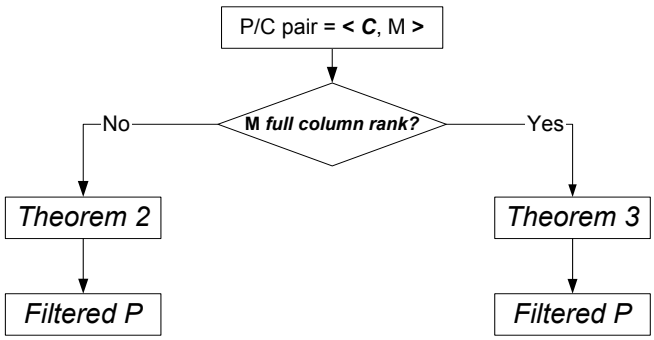


Figure 4.4: A hybrid approach that combines the ILP and the SNF approaches

It is possible, however, that despite having a non-full column rank dependency matrix, for some cases the SNF approach will generate a correct description of the producer domain. Detecting those cases would allow to extend the use of the SNF approach and express the producer domains in a more compact form than the one generated by the ILP approach.

4.4.1 The SNF & HNF guarded with ILP approach

In this section we present a procedure to determine *the structure of the producer lattice induced purely by the dependency mapping* M. The procedure determines whether an arbitrary IP from an output port is a hole (i.e., there is no corresponding consumption). This procedure is an extension of the procedure used in Section 4.3 for filtering holes arising due to mapping the consumer IS through a full column rank dependency matrix, and it makes use of the Smith and Hermite Normal Forms of this matrix. However, the procedure is not safe as holes can also be introduced in the presence of a non-full column rank matrix by the consumer domain constraints. Thus (as shown further), to ensure the correctness of the procedure, an ILP test has to be performed.

Consider the dependency function $f = Mx + N$, where M is an arbitrary integral matrix and N is an integral offset vector. Without loss of generality we assume $N = 0_n$. We want to determine

whether an arbitrary point y is an element of a given lattice :

$$L(M) = \{y \in Z^m \mid \exists x \ \in Z^n \ s.t. \ y = Mx\},$$

where M is an $(m \times n)$ integral matrix. Consider the example given in Figure 4.5 where the dependency function is expressed by the mapping matrix: $M = \left[\begin{smallmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \end{smallmatrix}\right]$. This matrix is not full column rank and therefore, we can not apply the SNF filtering approach.

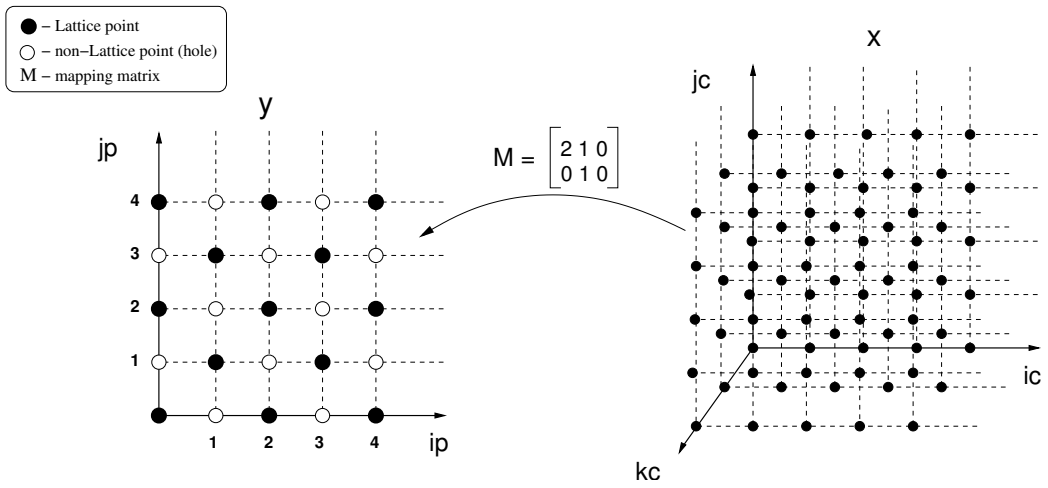


Figure 4.5: Lattice - regular holes introduced by the mapping

From Figure 4.5 we observe that without taking into account any of the consumer domain constraints, the non-full column rank matrix M induces a producer domain represented by the following lattice:

$$L(M) = \{(i_P, j_P) \in Z^m \mid (i_P - j_P)\%2 = 0\}. \tag{4.5}$$

Once we are able to derive such representation and confirm that there are no other holes (of different nature than induced by M) are presented at the producer side, the equivalent if-statement representation follows.

Solution

Deriving the OPD lattice explicit form:

To arrive at an explicit OPD lattice representation (as the one given in 4.5) we make use of the Hermite and Smith Decompositions of the dependency mapping matrix M. Notice that M can be non-full row rank $i.e., (rank(M) \leq m)$. In order to decide whether an arbitrary $y \in Z^m$ belongs to $L(M)$, we first derive the general form of the solution to the equation: $y = M \ x$. After that we impose that the solution x should belong to Z^n.

According to Smith Decomposition of a matrix M, there exist two unimodular matrices U and V such that:

$$U \ M \ V = S,$$

where U and V are $(m \times m)$ and $(n \times n)$ matrices, respectively, with $det(U) = -1$ and $det(V) = 1$. S is a $(m \times n)$ matrix of the form $S = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$, where D is a $(k \times k)$ diagonal matrix with diagonal entries $\delta_i \in Z_+ \setminus \{0\}$. This decomposition implies that:

$$M = U^{-1}\ S\ V^{-1}. \tag{4.6}$$

Substituting the expression of M from 4.6 into $y = M\ x$ we obtain that:

$$y = U^{-1}\ S\ V^{-1}\ x. \tag{4.7}$$

Because D is diagonal and V is invertible the following holds:

$$S\ V^{-1} = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}\ V^{-1} = \begin{bmatrix} L \\ 0 \end{bmatrix}, \tag{4.8}$$

where L is a full row rank $(k \times n)$ matrix. Substituting 4.8 in 4.7 we get:

$$y = U^{-1}\ \begin{bmatrix} L \\ 0 \end{bmatrix}\ x \Rightarrow U\ y = \begin{bmatrix} L \\ 0 \end{bmatrix}\ x.$$

This equation can be decomposed into two parts:

$$\begin{cases} U_1\ y = L\ x, \\ U_2\ y = 0, \end{cases}$$

where U_1 is a $(k \times m)$ matrix consisting of the first k rows of U and U_2 is a $((m-k) \times m)$ matrix consisting of the last $m-k$ rows of U. Thus, U is of the form: $U = \left[\begin{smallmatrix} U_1 \\ U_2 \end{smallmatrix}\right]$. Because L is a full row rank $(k \times n)$ matrix, it can be represented in the Hermite Normal Form. Therefore, there exists a unimodular $(n \times n)$ matrix C such that:

$$L\ C = [H\ 0], \tag{4.9}$$

where H is a $(k \times k)$ invertible matrix. Suppose $x = C\ x'$. Because C is unimodular $\Rightarrow x' \in Z^n$. Thus,

$$U_1\ y = L\ C\ x'. \tag{4.10}$$

By substituting 4.10 into 4.9 we get that:

$$U_1\ y = \begin{bmatrix} H & 0 \end{bmatrix}\ x',$$

which implies that:

$$U_1\ y = H\ x'_1,$$

where x'_1 consists of the first k elements of $x' = \left(\begin{smallmatrix} x'_1 \\ x'_2 \end{smallmatrix}\right)$. We observe that x'_2 can be any arbitrary vector in Z^{n-k}. Because H is invertible the following holds:

$$x'_1 = H^{-1}\ U_1\ y,$$

and because $x' \in Z^n$ we have that:

$$H^{-1}\ U_1\ y \in Z^k.$$

Thus, the following two relations represent the conditions that y has to respect in order to guarantee the existence of an integral solution to the equation $y = Mx$:

$$H^{-1}\ U_1\ y \in Z^k, \tag{4.11}$$

$$U_2\ y = 0. \tag{4.12}$$

Equation (4.12) shows that lattice L lays on a hyper-plane in Z^m and contains only points that respect (4.11). We observe here that (4.12) is expressed as a set of linear constraints, while (4.11) is expressed as a set of pseudo-linear constraints, e.g., modulo operands. We remark the necessity of generating the matrix H^{-1} (in order to derive the modulo operands used for testing (4.11)) as made of rational numbers expressed as fractions, where each such fraction has integer nominator and denominator.

ILP correctness test

Once we derived the expression of the lattice points induced by the matrix M, we can build the producer domain as follows:

$$P = \{y \in Z \mid y \in M(\mathcal{C}(N)),\ H^{-1}\ U_1\ y \in Z^k\}. \tag{4.13}$$

The condition (4.12) has been omitted because $y \in M(\mathcal{C}(N)) \Rightarrow U_2 y = 0$. However, to guarantee the correctness of the approach presented before we have to determine whether all the holes induced at the producer side are modeled by the constraints given by 4.11 and 4.12. If a hole is not modeled by those two constraints then we call it an *irregular hole*. To determine the existence of irregular holes we make use of the empty domain test (EDT) introduced in Section 2.3.4. For this purpose we build the set consisting of all the irregular holes, and then apply EDT that returns false in case the set is not empty. The set consisting of the irregular holes y is determined by the following constraints:

$$S = \begin{cases} x \in \mathcal{C}(N), & (c1) \\ y = M(x), & (c2) \\ y \in M(\mathcal{C}(N)), & (c3) \\ H^{-1}\ U_1\ y \notin Z^k. & (c4) \end{cases}$$

Observe that $(c4)$ does not represent a linear constraint. Nevertheless, it can be transformed into a series of linear constraints. Suppose that $H^{-1}\ U_1\ y = F(y)/d$, where $F(x)$ is a linear function with integer coefficients and d is an integer number. Hence,

$$H^{-1}\ U_1\ y \notin Z^n \equiv \exists\ k \in Z^n \text{ such that } \begin{cases} F(y) & = kd + 1, \\ & : \\ F(y) & = kd + d - 1. \end{cases}$$

Thus, $S = \bigcup_{i=1}^{d-1} S_i$, where S_i represents the following set:

$$S_i = \begin{cases} x_c \in \mathcal{C}(N), & (c1) \\ y_p = M(x_c), & (c2) \\ y_p \in M(\mathcal{C}(N)), & (c3) \\ F(y) = kd + i. & (c4') \end{cases}$$

Hence, in order to test the existence of irregular holes a number of $d-1$ empty domain tests (EDT) have to be solved:

$$EDT(S) = \bigvee_{i=1}^{d-1} EDT(S_i).$$

If all the tests are true then no irregular holes are found and therefore, the constraints given by (4.11) and (4.12) represent the producer lattice.

4.4.2 Restructuring the hybrid approach

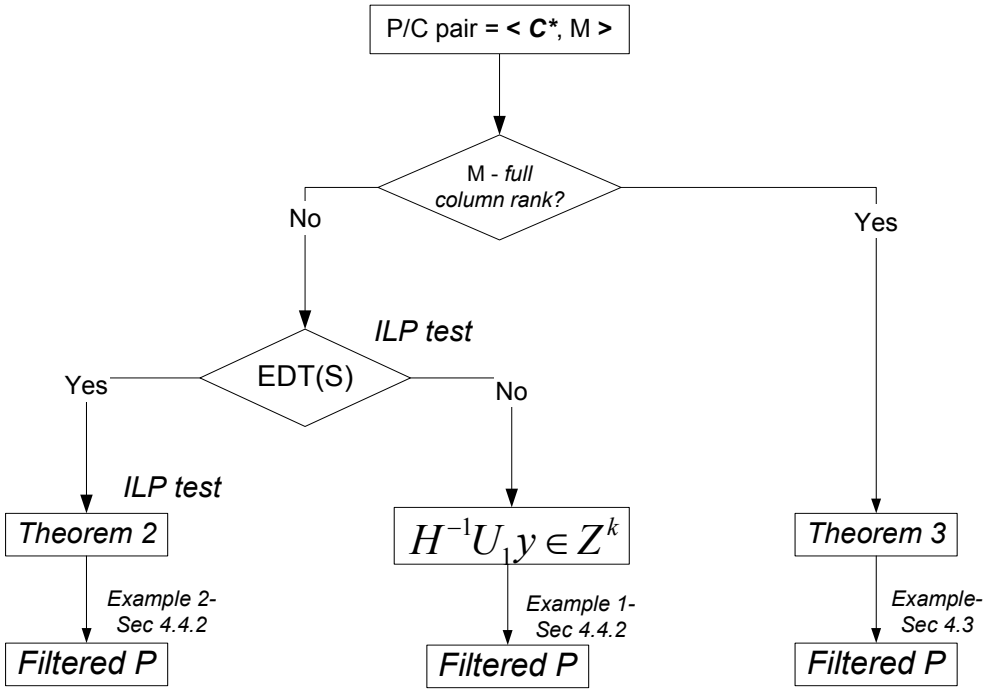


Figure 4.6: Restructuring the hybrid filtering approach

According to the previous section, we can express producer domains in a more compact form than by using purely the hybrid approach presented in Figure 4.4. Consequently, the hybrid approach is restructured, resulting in the decision tree shown in Figure 4.6. The restructured hybrid approach starts with checking whether the dependency matrix is of full column rank and in case yes, it derives the producer lattice based on Theorem 3. Otherwise, it checks whether irregular holes are present, and in case yes, applies the ILP test as stated by Theorem 2 from Section 4.2. Otherwise, the regular holes are derived using relation (4.11). Remark that the sign $*$ from the consumer $\mathcal{C}^*$ denotes that the consumer domain does not include equalities. This assumption is made based on Section 5.6.1, where a procedure for removing the equalities from the consumer side is presented.

We illustrate the decision paths depicted in Figure 4.6 with three corresponding examples. The case of the full column rank dependency matrix is illustrated by the example presented in Section 4.3. We further present two examples corresponding to the case in which the matrix is not full column rank.

Example 1: Here we consider the case where the previous approach based on matrix decomposition together with the ILP correctness test can be applied. Figure 4.7 illustrates the case in which the dependency is not full column rank and no irregular holes are induced at the producer side.

Producer:

```
for (int y1=0; y1<=2; y1++){
 for (int y2=0; y2<=4; y2++){
  for (int y3=0; y3<=3; y3++){
      r[y1,y2,y3]=Init(...);
  }
 }
}
```

Consumer:

```
for (int x1=0; x1<=1; x1++){
 for (int x2=0; x2<=1; x2++){
  for (int x3=0; x3<=3; x3++){
   Sink[k1,k2]=Fc( r[2*x1,4*x1,x3] );
  }
 }
}
```

Figure 4.7: P/C program before Production Restructuring

Figure 4.8 presents the ISs of Producer and Consumer, together with the dependency matrix M. What we have to do is to find the conditions under which an integral point y belongs to the lattice $L(M)$.

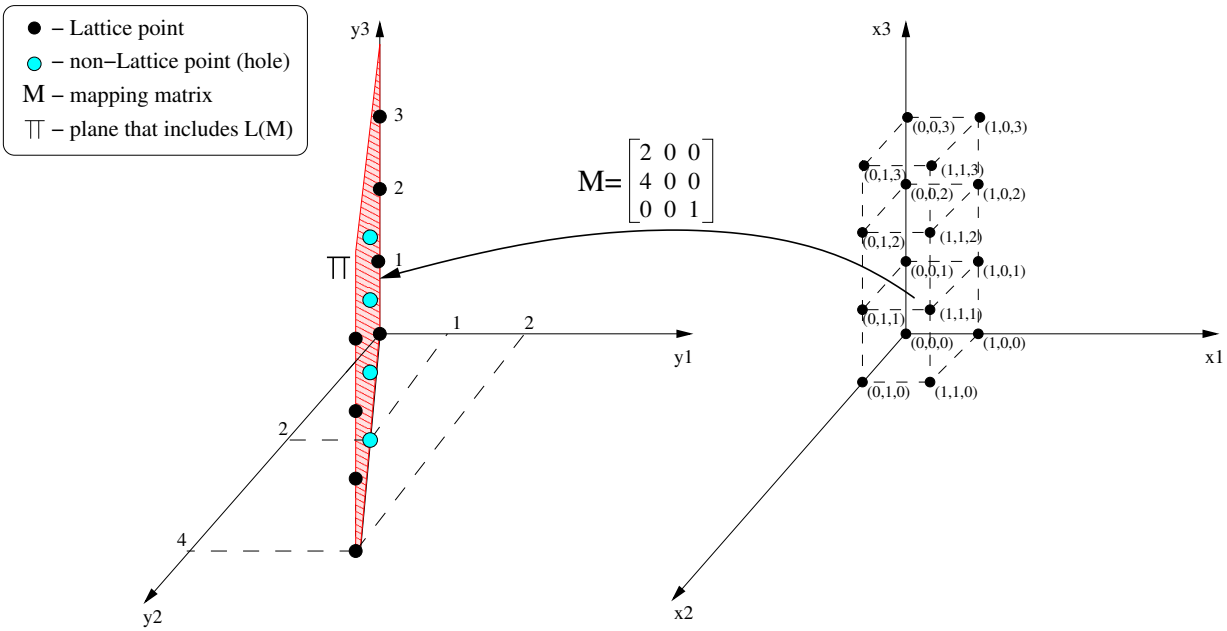


Figure 4.8: Inducing only regular holes through a non-full column rank matrix

Following the approach described before, we first use the Smith Decomposition of the mapping M, ($U\ M\ V = S$). According to this, we get the unimodular matrices U, V and the diagonal matrix S:

$$U = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 2 \\ -2 & 1 & 0 \end{bmatrix}, \quad U^{-1} = \begin{bmatrix} 2 & -1 & 0 \\ 4 & -2 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$V = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix}, \quad V^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 1 \end{bmatrix}.$$

From $L = S\ V^{-1}$, we get:

$$\begin{bmatrix} L \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ -2 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}.$$

Because L is a full row rank matrix, we can derive the Hermite Normal Form ($L\ C = [H, 0]$):

$$C = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix} \quad \Rightarrow [H, 0] = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \end{bmatrix}.$$

Now we compute H^{-1}, and multiply it with U_1, which consists of the first $k = 2$ rows of U:

$$H^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1/2 \end{bmatrix}, \quad U_1 = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 2 \end{bmatrix}.$$

The following holds:

$$H^{-1}\ U_1 = \begin{bmatrix} 0 & 0 & 1 \\ -1/2 & 0 & 0 \end{bmatrix}.$$

According to the condition expressed in (4.11) (i.e., $H^{-1}\ U_1\ y \in Z^k$) we find:

$$0\ y_1 + 0\ y_2 + 1\ y_3 \in Z, \tag{4.14}$$

$$-1/2\ y_1 + 0\ y_2 + 0\ y_3 \in Z. \tag{4.15}$$

According to the condition expressed in (4.12) (i.e., $U_2\ y = 0$) we find:

$$-2\ y_1 + 1\ y_2 + 0\ y_3 = 0.$$

Since $y = (y_1, y_2, y_3)$ is an integer point (IP from a *for-loop*) the condition expressed by (4.14) is automatically valid. In conclusion, an arbitrary integer point $y = (y_1, y_2, y_3)$ belongs to the lattice $L(M)$ if and only if:

$$-\ y_1/2 \in Z, \tag{4.16}$$

$$y_2 = 2\ y_1. \tag{4.17}$$

As shown in Figure 4.8, the equation (4.17) represents the plane Π on which the lattice L lays, while equation (4.16) filters out the points with odd y_1 coordinate.

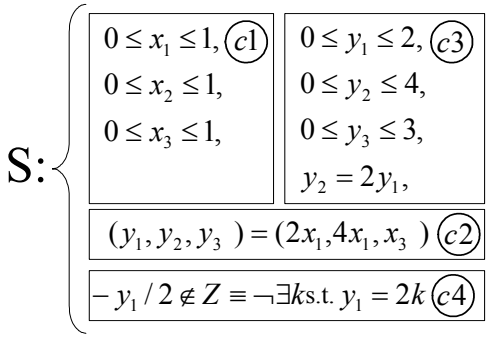


Figure 4.9: ILP correctness test

ILP correctness test: To guarantee that all the holes induced at the producer side are modeled by (4.16) and (4.17), we have to solve an ILP correctness test. This can be done by testing whether the domain given in Figure 4.9 contains integer points (i.e., irregular holes). Observe that the condition $(c4)$ from page 33 (i.e., $H^{-1}\ U_1\ y \notin Z^k$), has been translated to a test for the existence of an integer k such that $y_1 = 2k$. In our example, we get that $EDT(S) = true$, and therefore, as shown in the following piece of code, the producer domain is correctly filtered by the constraint $(-y_1)/2 \in Z$:

Producer:

```
for (int y1=0; y1<=2; y1++){
 for (int y2=0; y2<=4; y2++){
  for (int y3=0; y3<=3; y3++){
     if (y2==2*y1){
      if ((-y1)%2==0){
        r[y1,y2,y3]=Init(...);
      }
     }
  }
 }
}
```

Consumer:

```
for (int x1=0; x1<=1; x1++){
 for (int x2=0; x2<=1; x2++){
  for (int x3=0; x3<=3; x3++){
   Sink[k1,k2]=Fc( r[2*x1,4*x1,x3] );
  }
 }
}
```

Figure 4.10: Restructured data production

Example 2: Here we present an example where the above approach based on matrix decomposition cannot be applied, and therefore, the general approach based on integer linear programming must be used. In Figure 4.12, we present the ISs of Producer and Consumer together with the dependency matrix $S = [2, 1, 3]$. The last column of S represents the offset which does not play any role in the further discussion so we ignore it. The resulted matrix $S = [2, 1]$ is not of full column rank, so we first have to check whether irregular holes are introduced. We derive the expression $H^{-1}\ U_1\ y \notin Z^k$ and then build up the test for existence of irregular holes. We obtain that $U^{-1} = [1]$ and $V^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}$, which implies that $HU_1 = 1$. This implies that no regular holes are introduced by the mapping

Producer:
```
for (int j=0; j<=3*N+3; j++){
  r[j]=Init(...);
}
```

Consumer:
```
for (int k1=1; k1<=N; k1++){
 for (int k2=1; k2<=N; k2++){
  Sink[k1,k2]=Fc( r[2*k1+k2+3] );
 }
}
```

Figure 4.11: P/C program before Production Restructuring taken from [79]

$M = [2, 1]$. To check whether irregular holes are introduced, we set up an ILP correctness test as given in Section 4.4.1. Since no regular holes are introduced, the correctness test is identical with the one used to obtain the irregular holes specified by 4.20. As one can see in Figure 4.12, the Producer iteration $j = 17$ represents an irregular hole for which no corresponding consumption $(k_1, k_2) \in \mathcal{P} \cap Z^2$ exists. This abnormality cannot be detected by analyzing only the mapping matrix M, it depends also on the Consumer constraints.

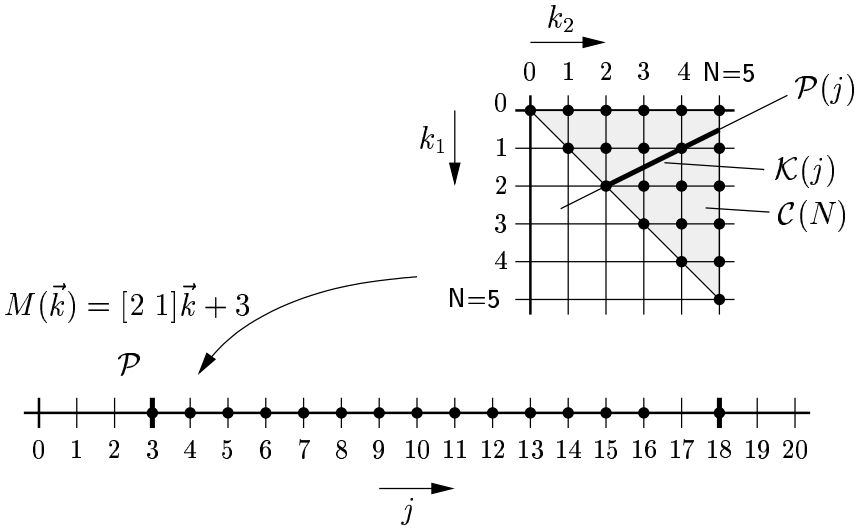


Figure 4.12: Inducing an irregular hole (i.e., $j = 17$) through a non-full column rank matrix

In the program depicted in Figure 4.11, the Consumer domain (see Figure 4.12) is described by the integral points inside the following parameterized polytope:

$$\mathcal{C}(N) = \{(k_1, k_2) \in R^2 \mid 0 \leq k_2 \leq N \wedge k_1 \leq k_2 \leq N\}. \tag{4.18}$$

From performing the real image of $\mathcal{C}(N)$ through M, (i.e., $M(\mathcal{C}(N))$) we obtain the following domain[6]:

$$\mathcal{P} = \{x \in R \mid 3 \leq x \leq 3N + 3\}. \tag{4.19}$$

[6]Note that $\mathcal{P}$ does not represent the Producer domain (OPD) which is in this case is a LBL bounded by $\mathcal{P}$.

To derive the Producer domain, we use $\mathcal{P}$ and M to set up the following PIP problem:

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq k_1 \leq N, \\ & k_1 \leq k_2 \leq N, \\ & 3 \leq \mathrm{j} \leq 3N+3, \\ & \mathrm{j} = 2k_1 + k_2 + 3, \\ \textbf{objective:} \quad & max_{\textbf{lex}}\{(k_1, k_2)\}. \end{aligned} \tag{4.20}$$

The solution of this PIP problem is given by a solution tree specified by the following piece of code:

```
1. if (0 <= j-3),
2.     if(0 <= -j+3*N+3),
3.       d18 = div(2*j,6),
4.           if(0 <= -j+N+2*d18+1),
5.               (k1,k2)=(d18-1,j-2*d18-1);
6.           else
7.               Nil;
8.           end
9.     else
10.       Nil;
11.    end
12. else
13.     Nil
14. end.
```

The solution tree has one non-empty branch which means that irregular holes are present. This branch leads to the one-dimensional LBL that represents the Producer domain as a two dimensional polytope with coordinates j and d_{18}:

$$\mathcal{P}(j, d_{18}) = \{(j, d_{18}) \in Z^2 \mid 3 \leq j \leq 3N+3 \quad \wedge \quad 0 \leq 2j - 6d_{18} \leq 5 \quad \wedge \quad 0 \leq -j + 2d_{18} + N + 1\}.$$

As shown in Figure 4.12, for $N = 5$ the Producer iteration $j = 17$ has to be filtered such that data will not be sent through the communication channel. The filtering is realized by adding an extra condition $0 \leq -j + N + 2 * d_{18} + 1$ to the conditions specified by (4.19). As a consequence, the P/C pair is restructured as shown in Figure 4.13. We observe that the condition $0 \leq 2j - 6d_{18} \leq 5$ is equivalent with the condition $d_{18} = div(2 * j, 6)$.

Producer:

```
for (int j=0; j<=3*N+3; j++){
 d18=div(2*j,6);
  if(0<=-j+2*d18+N+1){
    r[j]=Init(...);
  }
}
```

Consumer:

```
for (int k1=1; k1<=N; k1++){
 for (int k2=1; k2<=N; k2++){
  Sink[k1,k2]=Fc( a[2*k1+k2+3] );
 }
}
```

Figure 4.13: The refined P/C program, after performing Production Restructuring

4.5 Discussion

Theoretically, the Consumption Restructuring can be implemented independently and in a similar way as for the Production Restructuring. Below we show that such an implementation would be computationally more complex than the presented implementation as it requires extensive use of an ILP solver. To derive the OPD of an arbitrary producer statement P_j with respect to the consumption by statement C_i, we can make use of a formula similar to the one used to express an IPD in (3.4). Suppose there are multiple write accesses to the array r originating from a number of m assignment statements of which P_j, is a generic one. Thus, an arbitrary OPD of the producer statement P_j represents the IPs at which data is consumed by statement C_i and is expressed as follows:

$$\begin{aligned} OPD = \{x_p \in \mathcal{P}_j |\ & \exists\ y_c\ s.t.\ y_c \in \mathcal{C}_i \wedge x_p \prec y_c \wedge write_{Pj}(x_p) = read_{Ci}(y_c) \\ & \wedge (\ \forall k,\ 1 \leq k \leq m,\ \neg\ \exists\ x'_p \in \mathcal{P}_k\ s.t. \\ & x_p \prec x'_p \prec y_c\ \wedge\ write_{Pk}(x'_p) = read_{Ci}(y_c)\ )\ \}, \end{aligned}$$

where $\mathcal{P}_j$, $\mathcal{C}_i$ and $\mathcal{P}_k$ represent the iteration spaces of the statements P_j, C_i and P_k, respectively, and $write_{Pj}$, $write_{Pk}$, and $read_{Ci}$ represent the write accesses for the producer statements P_j and P_k, and the read access of the consumer statement C_i, respectively.

Chapter 5

Communication Model Selection

After Consumption and Production Restructuring, the storage structure of the original application is transformed such that each P/C pair of assignment statements communicates each argument over a dedicated multidimensional memory array as shown at the left in Figure 5.1.

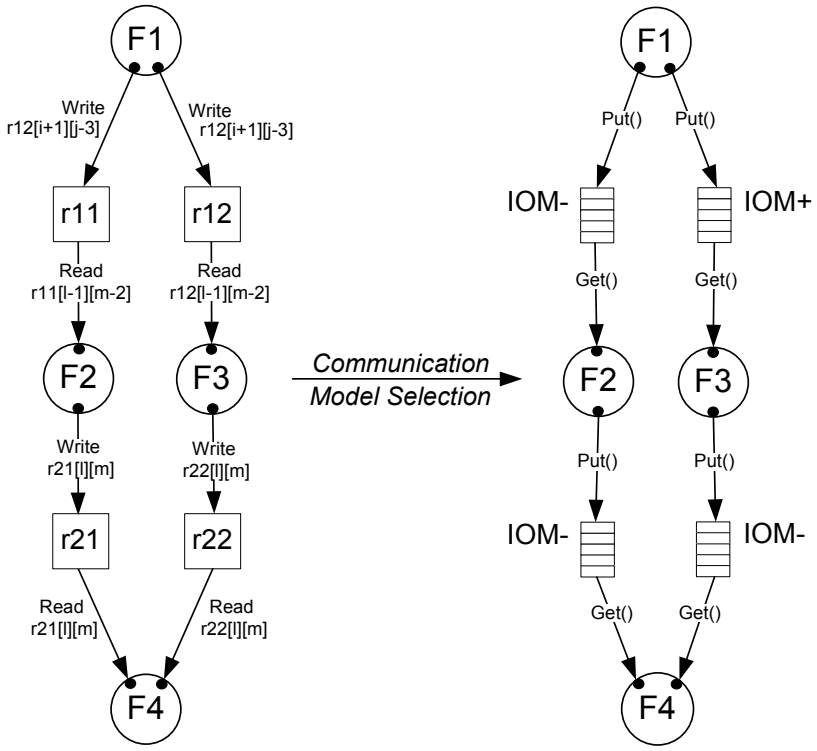


Figure 5.1: Communication Model Selection applied to the running example. See Section 5.1 for the meaning of IOM- and IOM+.

In general, it is desirable that a communication of an argument could occur via a FIFO channel instead of a multidimensional array because, for example, such communication model allows a simple implementation in hardware. However, due to the particular way in which data flows from a producer to a consumer, mapping array communication onto FIFO channels requires complex address generators, especially if the arrays have multiple dimensions. In the *Communication Model Selection* step, we

investigate the communication characteristics of each such P/C pair of assignment statements without changing the program structure. The presented compilation step addresses this problem, leading in most of the cases to the synthesis of a process network with a FIFO structure used to communicate data between the processes. In the case of our running example, the outcome of this step is graphically depicted at the right of Figure 5.1. As shown, the network communication structure is entirely represented by FIFO channels. Details are the subject of subsections 5.2.1 and 5.2.3.

5.1 Problem Definition

The communication of a P/C pair belongs to one of four types that are graphically illustrated in Figure 5.2. Those types are a result of the *ordering* of the iterations at the Producer and the Consumer side, and of *multiplicity* of a given token, which means that a token that is sent by the Producer is read more than once at the Consumer side.

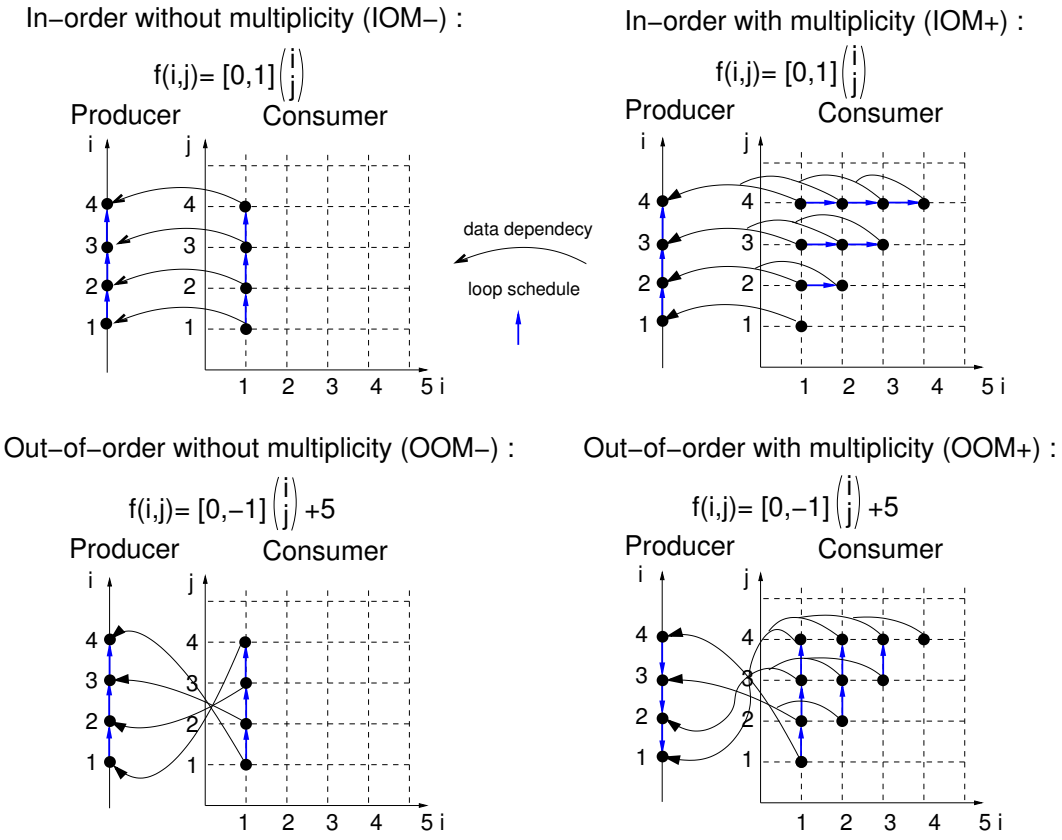


Figure 5.2: Four possible types of P/C data-flow graphs

The notions of ordering and multiplicity are formally defined as follows.

Definition 3 *A P/C pair is* **in-order** *iff the dependency function* $f : (\mathcal{C} \cap Z^n) \rightarrow P$ *preserves the token order, i.e., every two Consumer iteration points* $x_1 \prec x_2$ *are mapped onto two Producer iteration points* $y_1 = f(x_1)$ *and* $y_2 = f(x_2)$ *such that* $y_1 \preceq y_2$*. If a P/C pair is not in order we call it* **out-of-order**.

Definition 4 *A P/C pair is* **without multiplicity** *iff the mapping* $f : (\mathcal{C} \cap Z^n) \rightarrow P$ *is injective, i.e.,*$\forall\ x_1, x_2 \in \mathcal{C} \cap Z^n\ s.t.\ x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$. *Otherwise we say that the P/C pair is* **with multiplicity**.

According to these two definitions, an arbitrary P/C pair 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+)*.

As further discussed in Chapter 6, for each class an adequate communication mechanism is synthesized. Depending on the class, these communication mechanisms have significantly different implementation costs. Hence, from the performance/cost perspective, we are confronted with the problem of making an accurate classification for, and identification of the communication type of an arbitrary P/C pair.

5.2 Solution - ILP approach

To accurately determine the communication type of an arbitrary P/C pair, we rely on two tests. The *Reordering Test* (RT) determines whether a P/C pair is in-order or out-of-order, and the *Multiplicity Test* (MT) determines whether a P/C pair is with or without multiplicity. Based on these two tests, an arbitrary P/C pair is correctly classified to one of the previous four types. In the next sub-sections we show how these two tests can be set up using integer linear programming (ILP). For this purpose, the Multiplicity Test and the Reordering Test are formulated as empty domain tests (ET) i.e., tests (introduced in Section 2.3.4) that determine whether a domain specified by linear constraints contains at least one integral point. In case of the MT, we will refer to the underlying domain as to *Multiplicity Problem* and in case of RT as to *Reordering Problem*.

5.2.1 The Multiplicity Test

From Definition 4, we formalize the *Multiplicity Problem* (MP) as the following set:

$$\mathbf{MP}: \begin{cases} x \in (\mathcal{C}(N) \cap Z^k), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^k), & (c2) \\ x \neq y, & (c3) \\ f(x) = f(y). & (c4) \end{cases}$$

We say that a P/C pair has multiplicity if there exist an integral solution (x, y) to the corresponding MP. Thus, a P/C pair has multiplicity if there are two different Consumer points x and y that respect conditions $(c1)$, $(c2)$ and $(c3)$, such that they are mapped to the same point at the Producer side, as given by condition $(c4)$. We define the *multiplicity test* $MT(MP)$ to be a logical expression which is true if, and only if, there exists a solution to the multiplicity problem MP. Notice that $MT(MP) = true$ is equivalent to $ET(MP) = false$. The Multiplicity Test can be formalized in terms of integer linear programming.

Proposition 1 *The Multiplicity Test on a P/C pair with an n-dimensional Consumer iteration space, can be performed by solving at most $2n$ parametric integer linear programming problems.*

Proof: Consider that the dependency function f is represented by a matrix M. A possible offset does not play any role in the multiplicity problem, and therefore, we omit it.

Relation $(c3)$ implies that either $y \prec x$, or $x \prec y$, which makes allows the initial problem to be decomposed into two sub-problems called *Primal Multiplicity Problem* and *Dual Multiplicity Problem*:

$$\mathbf{PMP}: \begin{cases} x \in (\mathcal{C}(N) \cap Z^n), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^n), & (c2) \\ x \prec y, & (c3') \\ Mx = My. & (c4) \end{cases} \qquad \mathbf{DMP}: \begin{cases} x \in (\mathcal{C}(N) \cap Z^n), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^n), & (c2) \\ x \succ y, & (c3'') \\ Mx = My. & (c4) \end{cases}$$

If any of those problems contains integer points, then *MP* is true and, consequently, *ET(MT)* is false. Hence,

$$ET(MP) = ET(PMP) \quad \wedge \quad ET(DMP). \tag{5.1}$$

If MP has a solution, then it also has the lexicographically greatest solution. Finding the lexicographically greatest iteration y for MP can be formulated as a *pseudo multidimensional knapsack* problem (where (x, y) is a variable and n is a parameter). We say "pseudo knapsack problem" because the relations $(c3')$ and $(c3'')$ are not linear inequalities. However, they can be decomposed into sets of linear inequalities. Thus, each of the previous problems will be formulated as a set of parametric integer linear programming problems.

Consider that the Consumer domain is embedded in an n-dimensional iteration space such that $x = (\mathsf{x}_1, \mathsf{x}_2, ..., \mathsf{x}_n)^T$ and $y = (y_1, y_2, ..., y_n)^T$, where the superscript ^{T} denotes the transposition of the (row) vectors x and y. Then we can formulate *PMP* and *DMP* as unions of of parametric ILP problems.

$$PMP = \bigcup_{i=1}^{n} PMP_i, \tag{5.2}$$

where PMP_i, $i = 1, \ldots, n$ is given by

$$\mathbf{PMP_i}: \begin{cases} x = (\mathsf{x}_1, \mathsf{x}_2, ..., \mathsf{x}_n)^T \in (\mathcal{C}(N) \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (\mathcal{C}(N) \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (\mathsf{x}_1, ..., \mathsf{x}_{i-1}), \\ x_i < \mathsf{y}_i, \end{cases} \\ Mx = My. \end{cases}$$

Similarly,

$$DMP = \bigcup_{i=1}^{n} PMP_i, \tag{5.3}$$

where $DMP_i, i = 1, \ldots, n$ is given by:

$$\mathbf{DMP_i}: \begin{cases} x = (\mathsf{x}_1, \mathsf{x}_2, ..., \mathsf{x}_n)^T \in (\mathcal{C}(N) \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (\mathcal{C}(N) \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (\mathsf{x}_1, ..., \mathsf{x}_{i-1}), \\ x_i > \mathsf{y}_i, \end{cases} \\ Mx = My. \end{cases}$$

From relations 5.1, 5.2 and 5.3 we have,

$$ET(MP) = \bigwedge_{i=1}^{n} ET(PMP_i) \ \wedge \ \bigwedge_{i=1}^{n} ET(DMP_i). \tag{5.4}$$

However, since $DMP_i = U(PMP_i)$, where $U = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ is $2n$-dimensional matrix which exchanges the coordinates x and y. Obviously, $(\mathsf{x}, y) \in PMP_i \Leftrightarrow (y, \mathsf{x}) \in DMP_i$ which implies that $ET(PMP_i) = ET(DMP_i)$. Thus, to decide whether a P/C pair is with multiplicity it is enough to perform ET for only one of the problems PMP or DMP. Hence, at most n parametric ILP problems have to be solved for MT. □

As explained in Section 2.3, the solution of a PIP problem is represented by a multistage conditional expression,

$$\begin{aligned} &\text{if } (\mathsf{N} \in \ \mathcal{P}_1), \\ &\qquad\qquad \text{then} \quad (x, y) = (a_1, b_1), \\ &\text{if } (\mathsf{N} \in \ \mathcal{P}_2), \\ &\qquad\qquad \text{then} \quad (x, y) = (a_2, b_2), \\ &\qquad\qquad \vdots \\ &\text{if } (\mathsf{N} \in \ \mathcal{P}_m), \\ &\qquad\qquad \text{then} \quad (x, y) = (a_m, b_m), \end{aligned}$$

where $\mathcal{P}_1, ..., \mathcal{P}_m$ are polytopes. Thus, if all PMP_i and DMP_i are void of integral points (i.e., $(a_j, b_j) = (\perp, \perp), \forall j = 1...m$), then the correspondent P/C pair is without multiplicity. Otherwise, the P/C is with multiplicity.

Running example

Next we present how MT is applied to our running example. The results are shown in Fig 5.1. To decide on the multiplicity characteristics of the pair $PC_4 = (F1, F3)$, we verify whether the domain specified by the constraints MP_{PC4} contains integral points. All the constraints in MP_{PC4} are linear inequalities excepting those specified by the condition $(c3)$. x and y represent arbitrarily points from $\mathcal{C}(N)$.

$$\mathrm{MP}_{\mathrm{PC4}}: \begin{cases} \boxed{\begin{array}{l} 1 \le x_t \le P, \ (c1) \\ 1 \le x_l \le M, \\ 4 \le x_m \le N, \\ 8 \le x_l + x_m, \end{array}} \quad \boxed{\begin{array}{l} 1 \le y_t \le P, \ (c2) \\ 1 \le y_l \le M, \\ 4 \le y_m \le N, \\ 8 \le y_l + y_m, \end{array}} \\ \boxed{(x_t, x_l, x_m) \ne (y_t, y_l, y_m) \ (c3)} \\ \boxed{(x_t, x_l - 1, N) = (y_t, y_l - 1, N) \ (c4)} \end{cases}$$

Using the lexicographic order, the condition $(c3)$ is decomposed into a set of linear inequalities:

$$(x_t, x_l, x_m) \ne (y_t, y_l, y_m) \equiv$$

$$\begin{cases} x_t < y_t \vee & (c3^{P1}) \\ (\ x_t = y_t, x_l < y_l\ ) \vee & (c3^{P2}) \\ (\ x_t = y_t, x_l = y_l, x_m < y_m\ ) \vee & (c3^{P3}) \end{cases} \bigvee \begin{cases} x_t > y_t \vee & (c3^{D1}) \\ (\ x_t = y_t, x_l > y_l\ ) \vee & (c3^{D2}) \\ (\ x_t = y_t, x_l = y_l, x_m > y_m\ ). & (c3^{D3}) \end{cases}$$

This leads to six instances of MP, namely PMP_{1-3} and DMP_{1-3}. If one of these systems has a solution, multiplicity is involved. However, according to Proposition 1, it is enough to test PMP_{1-3}. The problem PMP_3 with conditions $(c1), (c2), (c3^{P3})$,and $(c4)$ contains integral points, which leads to the conclusion that the pair PC_4 is with multiplicity. This can be verified by looking, for example, to the points $P1 = (t, l, v)$ and $P2 = (t, l, w)$, with $v < w$. Both points are mapped to the same point $(t, l-1, N)$ at the Producer side. Similarly, by applying the multiplicity tests to the other 3 P/C pairs of our running example, we find that PC_1, PC_2, PC_3 are without multiplicity (M-).

5.2.2 Optimizations

In this section we present two optimizations which allow to reduce the number of ETs that need to be evaluated in order to solve the multiplicity problem.

Optimization 1

Observe that if $i < j$, then, most likely, the volume of the domain PMP_i is larger than the volume of the domain PMP_j because PMP_i contains less equalities than PMP_j. Thus, there is a higher chance that the domain PMP_i will contain integral points.

For example, consider that Consumer domain that is contained in a 2-dimensional iteration space with a dependency function f represented by the matrix $M = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Thus, the first place where to look for a solution is on the surface (PMP_1). Then comes the line (PMP_2). Figure 5.3 depicts graphical representation of the different domains in which to search for a solution for a given 2-dimensional iteration space. The same strategy is used for an n-dimensional iteration space. We first look for a solution in an as broad space as possible, and from there we narrow down the search space until we reach the line depicted in Figure 5.3 as PMP_2.

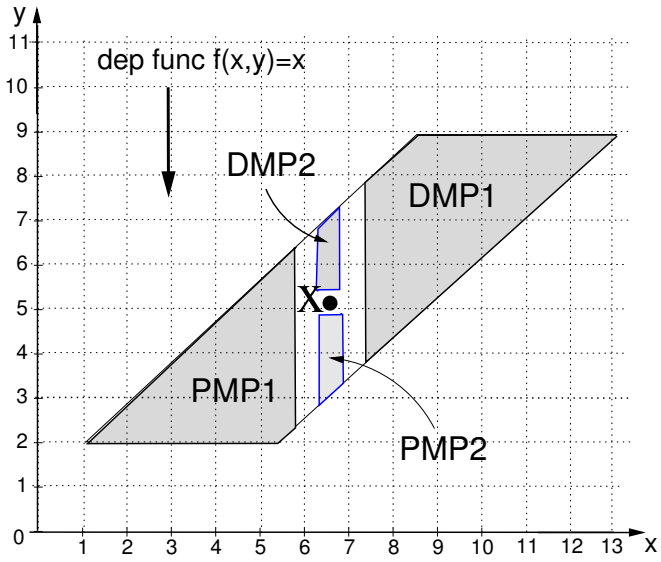


Figure 5.3: Decomposition of a domain into multiple multiplicity problems

Optimization 2

To further reduce the number of ETs that have to evaluate to solve the multiplicity problem, we make use of the SNF of the dependency matrix M.

Proposition 2 *Consider a P/C pair determined by a fully dimensional Consumer domain C and an $m \times n$ mapping matrix M. Suppose the SNF of M has t zero columns. Then, the multiplicity test can be formulated as t valid PIP problems, i.e., the other $n - t$ problems are always empty.*

Proof: According to the Smith Normal Form (see Chapter 2): $M = U^{-1} \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} V^{-1}$, where U and V are unimodular matrices. Because U and V are unimodular transformations it turns out that $Mx = My \Leftrightarrow [D, 0]\ x = [D, 0]\ y$.

Let C' be the real image of the domain C through the unimodular mapping V^{-1}. As a consequence, $M|_{(C \cap Z^n)}$ is injective iff $[D, 0]|_{(C' \cap Z^n)}$ is injective, such that applying MT restricted to the domain C under the mapping M is equivalent with applying MT restricted to the domain C' under the mapping $[D, 0]$. Analyzing a running primary multiplicity problem PMP_i:

$$\mathbf{PMP_i}: \begin{cases} x = (\mathsf{x}_1, \mathsf{x}_2, ..., \mathsf{x}_n)^T \in (C' \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (C' \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (\mathsf{x}_1, ..., \mathsf{x}_{i-1}), \\ x_i < \mathsf{y}_i, \end{cases} \\ [D, 0]\ x = [D, 0]\ y, \end{cases}$$

it turns out that $\forall i = 1...n - t$ the condition $(c4)$ is always false and therefore, $PMP_i = \emptyset$. Hence, $\forall i = 1...n - t, \quad ET(PMP_i) = true.$ □

Example

In the following example we illustrate how Optimization 2 can be applied to reduce the number of parametric ILP problems when determining the multiplicity characteristics of a P/C. Consider a P/C pair as specified after the Consumption and Production restructuring steps:

Producer:
```
for (int i=6-T; i <= 10; i++){
     a[ i ] = F1();
}
```

Consumer:
```
for (int i=6-T; i <= 10; i++){
  for (int j=2; j <= N; j++){
    if ( i+2 <= 3*j <= i+T ){
      Fc( a[ i ] );
    }
  }
}
```

Figure 5.4: A simple P/C pair

For this example, the P/C dependence graph is shown in Figure 5.5. Observe that the dependency function is represented by a matrix $M = [1, 0]$ which is already in SNF format. In general, we would

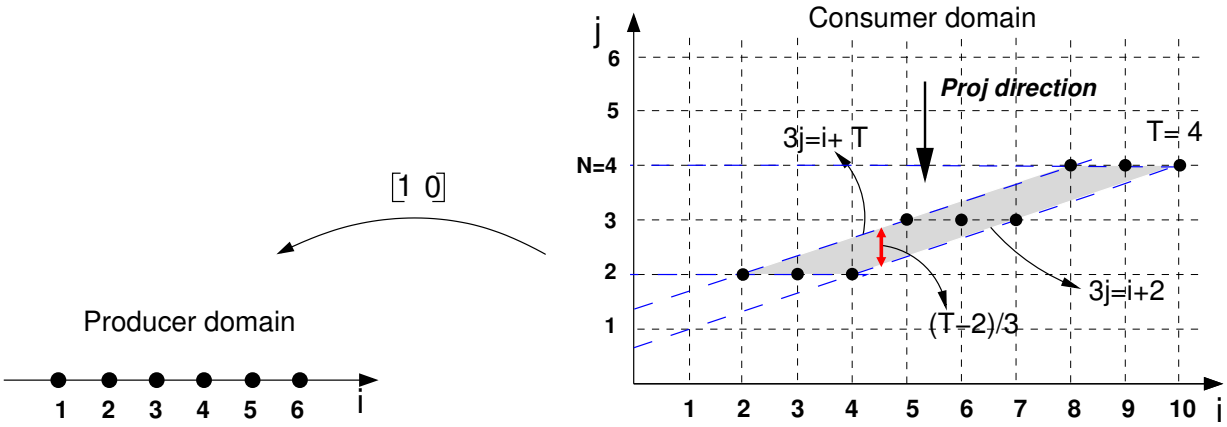


Figure 5.5: Injectivity depending on the parameters

have to consider the following two ILP problems:

$$\mathbf{PMP_1}: \begin{cases} x_1 + 2 \leq 3x_2 \leq x_1 + T, \\ 2 \leq x_2 \leq N, \\ y_1 + 2 \leq 3y_2 \leq y_1 + T, \\ 2 \leq y_2 \leq N, \\ x_1 < y_1 \\ x_1 = y_1. \end{cases} \quad \mathbf{PMP_2}: \begin{cases} x_1 + 2 \leq 3x_2 \leq x_1 + T, \\ 2 \leq x_2 \leq N, \\ y_1 + 2 \leq 3y_2 \leq y_1 + T, \\ 2 \leq y_2 \leq N, \\ x_1 = y_1, \\ x_2 < y_2, \\ x_1 = y_1. \end{cases}$$

However, using Optimization 2 it turns out that it is enough to perform $ET(PMP_2)$. As a result,

the following solution tree is obtained:

$$\begin{array}{l} \text{if } (5 <= T), \\ \qquad\qquad (x_1, x_2) = (y_1, y_2 - 1); \\ \text{else} \\ \qquad\qquad \perp . \end{array}$$

Discussion: In general, the multiplicity characteristic of a P/C pair may depend on the range of the parameters occurring in the specification of the consumer domain. As shown in the example, if $T < 5$, then the pair is without multiplicity. However, in our current compiler implementation we don't take into account the range of the consumer parameters. Thus, regardless the value of T, the P/C pair from this example will be considered as with multiplicity, and the corresponding communication mechanism will be applied.

5.2.3 The Reordering Test

The *Reordering Test* is used to determine whether the order of producing tokens at the Producer side is the same as the order in which they are consumed at the Consumer side. According to Definition 3, a P/C pair is out-of-order if two different Consumer points x and y (see conditions $(c1)$, $(c2)$) if $x \prec y$ $((c3))$ exist such that $f(y) \prec f(x)$ $((c4))$. These conditions can be captured in the following system specifying the *Reordering Problem* (RP):

$$\mathbf{RP}: \quad \begin{cases} x \in (\mathcal{C}(N) \cap Z^n), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^n), & (c2) \\ x \prec y, & (c3) \\ f(y) \prec f(x). & (c4) \end{cases}$$

Similar with the MT, the problem of deciding whether reordering occurs in a P/C pair is reduced to testing the existence of a solution for the RP.

Proposition 3 *The Reordering Test on a P/C pair with an n-dimensional Consumer iteration space and a $(n \times k)$ dependency matrix M, can be answered by solving $n \times k$ parametric integer linear programming problems.*

Proof: The lexicographic order " $\prec$ " can be rewritten using the following decomposition:

$$x \prec y \Leftrightarrow (x \prec_1 y) \vee (x \prec_2 y) \vee ... \vee (x \prec_n y), \tag{5.5}$$

where, $x \prec_i y$ means that $x_1 = y_1, x_2 = y_2, ..., x_{i-1} = y_{i-1}, x_i < y_i$.

According to the decomposition of the lexicographic order, the conditions $(c3)$ and $(c4)$ from the reordering problem are decomposed into linear constraints that define a number of disjoint sets. ET is then applied to each set. Suppose the Consumer domain is embedded in an n-dimensional iteration space. Let $x = (x_1, x_2, ..., x_n)^T$ and $y = (y_1, y_2, ..., y_n)^T$ be two different points from the Consumer domain. The dependency function f is represented by a $(n \times k)$ matrix M made of n rows :

$M = \begin{bmatrix} M_1 \\ \vdots \\ M_n \end{bmatrix}$. The RP is decomposed into $(n \times k)$ disjoint subproblems given as RP_{ij}, where $i = 1...n$ and $j = 1...k$:

$$\mathbf{RP_{ij}}: \begin{cases} x = (\mathsf{x}_1, \mathsf{x}_2, ..., \mathsf{x}_n)^T \in (C \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (C \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (\mathsf{x}_1, ..., \mathsf{x}_{i-1}), \\ \mathsf{x}_i < y_i, \\ (M_1 x, ...M_{j-1} x) = (M_1 y, ..., M_{j-1} y), \\ M_j y > M_j x, \end{cases} \end{cases}$$

such that:

$$\mathsf{ET(RP)} = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \mathsf{ET}(\mathsf{RP}_{ij}).$$

This means that if one of these systems has a solution, then re-ordering is involved. □

Running example

Consider again the P/C pair $PC_4 = (F1, F3)$ in our running example. To perform the Reordering Test (RT), we have to check whether the domain RP_{PC4} contains integral points:

$$\mathrm{RP}_{\mathrm{PC4}}: \begin{cases} \boxed{\begin{array}{l} 1 \le x_t \le P, \ (c1) \\ 1 \le x_l \le M, \\ 4 \le x_m \le N, \\ 8 \le x_l + x_m, \end{array}} \quad \boxed{\begin{array}{llll} 1 & y_t & P, & (c2) \\ 1 & y_l & M, & \\ 4 & y_m & N, & \\ 8 & y_l & y_m, & \end{array}} \\ \boxed{(x_t, x_l, x_m) \prec (y_t, y_l, y_m) \quad (c3)} \\ \boxed{(x_t, x_l - 1, N) \succ (y_t, y_l - 1, N) \ (c4)} \end{cases}$$

Conditions $(c3)$ and $(c4)$ contradict each other, and therefore, the reordering problem is without solution: PC_4 is in-order. By applying the reordering tests to the other three P/C pairs in the example, we find that PC_1, PC_2, PC_3 are all in order, as shown in Figure 5.1.

5.2.4 Optimization

Similarly to the first MT optimization, the number of ILP problems that have to be considered for RT, can be reduced. Observe that, most likely, the volume of the domain in problem RP_{ij} will be larger than the volume of the domain RP_{lm} when $(l, m) \prec (i, j)$. Thus, there is a higher chance that the domain RP_{ij} will contain integer points rather than RP_{lm}. Hence, we solve the following sequence of reordering problems: $\mathrm{RP}_{1,1}...\mathrm{RP}_{1,k}...\mathrm{RP}_{n,1}...\mathrm{RP}_{n,k}$. As soon as a solution is found, the search is stopped.

Example

In this example we show how the optimization can be applied to reduce the number of ILP problems. The example also shows that the reordering characteristics depend on parameters N and T occuring in the producer and consumer domains.

Producer:

```
for (int i=1; i <= N; i++) {
    for (int j=1; j <= N; j++) {
       if ( 4i-T <= 2j <= 4i-2 ) {
              a[ i, j ] = Fp();
       }
    }
 }
```

Consumer:

```
for (int i=1; i <= N; i++) {
   for (int j=1; j <= N; j++) {
      if ( i+1 <= 2j <= i+T ) {
            Fc( a[ j, i ] );
      }
   }
}
```

Figure 5.6: A simple P/C pair

Consider the P/C pair from Figure 5.6 with the dependency matrix $M = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$. The graphical representation of the P/C dependence graph is shown in Figure 5.7 for $T = 3$. Obviously there is no reordering involved: the order in which the iteration points are visited is the same for both Producer and Consumer: A, B, C, D, E, F, G and H.

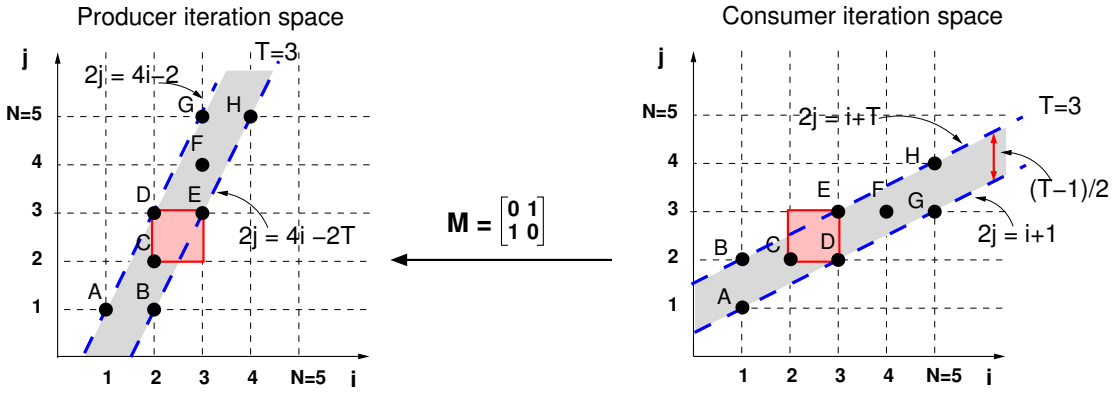


Figure 5.7: Reordering example with $T = 3$

However, as shown in Figure 5.8, for values of the parameter $T > 3$, the P/C pair becomes out-of-order. In this case, a unit square C, D, E, F embedded in the Consumer iteration space will be mapped into the unit square C, E, D, F at the Producer side. Thus, for the Consumer iteration space, the order the iteration points are visited is A, B, C, D, E, F, G, and H, while for the Producer, the order of the corresponding points through the mapping $M = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, is A, B, C, E, D, F, G, and H.

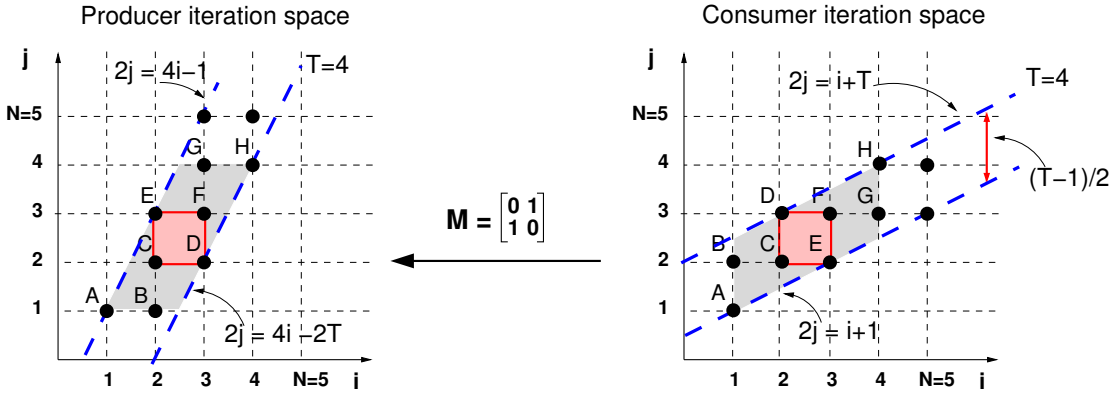


Figure 5.8: Reordering example with $T = 4$

In general we would have to solve the following 4 ILP problems:

$$\mathbf{PRP_{11}}: \begin{cases} x_1 + 1 \le 2x_2 \le x_1 + T, \\ 1 \le x_1 \le 4, \\ y_1 + 1 \le 2y_2 \le y_1 + T, \\ 1 \le y_1 \le 4, \\ x_1 < y_1, \\ M_1 x > M_1 y, \end{cases} \qquad \mathbf{PRP_{12}}: \begin{cases} x_1 + 1 \le 2x_2 \le x_1 + T, \\ 1 \le x_1 \le 4, \\ y_1 + 1 \le 2y_2 \le y_1 + T, \\ 1 \le y_1 \le 4, \\ x_1 < y_1, \\ M_1 x = M_1 y, \\ M_2 x > M_2 y, \end{cases}$$

$$\mathbf{PRP_{21}}: \begin{cases} x_1 + 1 \le 2x_2 \le x_1 + T, \\ 1 \le x_1 \le 4, \\ y_1 + 1 \le 2y_2 \le y_1 + T, \\ 1 \le y_1 \le 4, \\ x_1 = y_1, \\ x_2 < y_2, \\ M_1 x > M_1 y, \end{cases} \qquad \mathbf{PRP_{22}}: \begin{cases} x_1 + 1 \le 2x_2 \le x_1 + T, \\ 1 \le x_1 \le 4, \\ y_1 + 1 \le 2y_2 \le y_1 + T, \\ 1 \le y_1 \le 4, \\ x_1 = y_1, \\ x_2 < y_2, \\ M_1 x = M_1 y, \\ M_2 x > M_2 y. \end{cases}$$

Applying the optimization, ET has to be performed on the following problems: PRP_{11},PRP_{12}, PRP_{21} and PRP_{22}. The first problem does not have a solution. However, the second reordering problem has a solution specified by the following decision tree:

$$\begin{aligned} &\text{if } (3 < T), \\ &\qquad (x_1, x_2) = (y_1, y_2 - 1); \\ &\text{else} \\ &\qquad \bot\,. \end{aligned}$$

Therefore, we do not have to investigate PRP_{21} and PRP_{22}. As shown in Figure 5.8, it turns out that if $T > 3$, the order of producing data is different from the order of consuming data. Nevertheless, as we already mentioned in the case of MT, the range of domain parameters in not taken into account such that the considered P/C pair is out of order.

5.3 Experimental Results

Table 5.1 summarizes the results for eight applications in the application domain of imaging and signal processing. The first column in the table gives the number of P/C pairs that appear in a particular algorithm. This is followed by the number of Empty Domain tests needed in the Reordering Test and the Multiplicity Test to classify these P/C pairs. The numbers for the MT are obtained without using the proposed optimization procedure. The last four columns give the four different communication types and how many of the P/C pairs belong to that type. Using functional simulation, we verified for each algorithm that the classification is indeed correctly.

Algorithm	P/C pairs	RT	MT	IOM-	IOM+	OOM-	OOM+
QR-Decomp	12	73	30	12	0	0	0
SVD	118	1283	565	84	4	30	0
Faddeev	28	205	78	24	3	1	0
Gauss-Elimin	11	17	6	7	0	1	3
BeamFormer	98	408	196	88	4	6	0
Motion Estim	98	882	294	98	0	0	0
M-JPEG	50	178	93	33	17	0	0
Stereo Vision	173	1470	518	172	0	1	0

Table 5.1: Experimental results: classification of the edges in real-life examples to one of the four communication types using the Reordering Test and the Multiplicity Test.

Table 5.1 clearly shows that the most frequent communication type is IOM- (80%) and the least frequent one is OOM+ (1%). This is nice, as type IOM- requires only a simple FIFO buffer for communication. The OOM+ requires also re-ordering memory, a life-time controller, and a reordering controller, but hardly appears in networks. The second most frequent type is IOM+ (10%). This is also nice, as only a FIFO buffer together with some simple additional control to keep track of the life time of a token is needed for communication. Together with type IOM-, we can say that in 90% of the cases, a FIFO buffer can indeed be used to implement the communication between a P/C pair of statements.

5.4 Alternative Polynomial-time Classification Techniques

The two tests that we have presented so far (MT and RT) can provide an exact classification of the communication type for arbitrary P/C pairs. However, the following reasons force us to seek for alternative approaches:

- Solved as integer linear programming problems, the MT and RT are expensive in terms of computation time (NP complete) and memory resources.
- A typical signal processing algorithm like SVD contains a large number of P/C pairs, such that a large number of ILP systems (≈ 1800) with many variables (≈ 15) need to be solved.

Instead of solving ILP systems, we can exploit particular characteristics of the dependency matrix to classify a P/C pair to one of the four types.

For example, consider the example given in Figure 5.9. The two functions (f_p and f_c) and the

```
for (int i=1; i<=N, i++){
  for (int j=1; j<=N, j++){
    a[i,j] = fp();
  }
}
```

```
for (int x=1; x<=N, x++){
  for (int y=1; y<=N, y++){
    fc(a[x,y]);
  }
}
```

Figure 5.9: An In-order P/C pair

static array a[] define a P/C pair. The order of producing and consuming data are exactly the same and the mapping matrix is: $M = [\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}]$. Suppose now that we modify the Consumer such that it consumes tokens in a different order than they are produced.

```
for (int i=1; i<=N, i++){
  for (int j=1; j<=N, j++){
    a[i,j] = fp();
  }
}
```

```
for (int x=1; x<=N, x++){
  for (int y=1; y<=N, y++){
    fc(a[y,x]);
  }
}
```

Figure 5.10: An Out-of-order P/C pair

This can be achieved by interchanging the indices of the array a (from a[x,y] to a[y,x]) at the Consumer as shown in Figure 5.10. In this case the Consumer domain is the same as before, while the index change is reflected only in the affine mapping matrix that has become $M = [\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}]$. Apparently, the order change from an in-order case to an out-of-order case is reflected by the mapping matrix. Hence, by analyzing the mapping between Producer and Consumer, we want to decide whether out-of-order communication appears or not. Similarly, we want to say whether multiplicity is involved based on the structure of the mapping matrix. For that purpose, we introduce three alternative techniques to classify a P/C pair. The advantage of these techniques is that they have polynomial complexity, making them more suitable for a compiler implementation.

5.5 Order and Multiplicity detection using the P/C mapping matrix

In this section we analyze the communication type of a P/C pair, based only on the mapping matrix properties. We will propose a number of properties that allow us to classify a P/C pair based on the mapping matrix.

5.5.1 Order

We show that certain properties of the mapping matrix should be verified in order to take a decision regarding the communication type. Here we introduce the definition of an *in-order matrix* which helps us to formulate the reordering problem:

Definition 5 *We say that a $(m \times n)$ integral matrix M is* **in-order** *if the $(k \times n)$ matrix T made of the independent rows of M in the order they appear from top to bottom, is such that $t_{i,i} > 0$ for all i, and $t_{i,j} = 0$ for $i < j$. A matrix that is not in-order, we call out-of-order.*

The next proposition gives a necessary condition for communication without reordering:

Proposition 4 *If a P/C mapping matrix is in-order then the corresponding P/C pair is in order.*

Proof: Consider an in-order $(m \times n)$ integral matrix M and let be two different points $X = (x_1, ..., x_n)^T$ and $Y = (y_1, ..., y_n)^T$ such that $X \prec Y$; i.e., there exists an integer l, with $1 \leq l < n$ such that $\forall s$ such that $1 \leq s \leq l$ it follows that $x_s = y_s$ and $x_{l+1} < y_{l+1}$. According to Definition 5, the $(k \times n)$ matrix T made of the independent rows of M taken in the order they appear in the format of M, has the next shape:

$$\mathbf{T} = \begin{bmatrix} t_{11} & 0 & \dots & 0 & 0 & \dots & 0 \\ t_{21} & t_{22} & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ t_{k1} & t_{k2} & \dots & t_{kk} & 0 & \dots & 0 \end{bmatrix}.$$

By mapping X and Y through the matrix T, we get the points $A^{'} = T\ X = (a'_1, a'_2, ..., a'_k)^T$, respectively $B^{'} = T\ Y = (b'_1, b'_2, ..., b'_k)^T$, where $a'_i = \sum_{j=1}^{i} t_{ij} x_j$ and $b'_i = \sum_{j=1}^{i} t_{ij} y_j$. Now, depending on the value of k two cases can be distinguished:

- **Case1:** $k \leq l$ **:** Because $\forall j \leq l$, $x_j = y_j \ \Rightarrow \forall i = \overline{1,k}$, $a'_i = b'_i$, such that $A^{'} = B^{'}$. Because T is made of the independent rows of M, there is a $(m \times k)$ matrix L such that $M = L\ T$. But, $M\ X = L\ T\ X = L\ A'$ and $M\ Y = L\ T\ Y = L\ B'$. Hence $M\ X = M\ Y$.
- **Case2:** $l < k$ **:** Similarly to *Case 1*, $\forall i = \overline{1,l}$, $a'_i = b'_i$. But $a'_{l+1} = \sum_{j=1}^{l} t_{lj} \cdot x_j + t_{(l+1)j} \cdot x_{l+1}$ and $b'_{l+1} = \sum_{j=1}^{l} t_{lj} \cdot y_j + t_{(l+1)j} \cdot y_{l+1}$, and therefore, $a'_{l+1} < b'_{l+1}$. Using the definition of the lexicographical order, we get that $A^{'} \prec_{l+1} B^{'}$[1]. Suppose that row $l+1$ in T (for sure this row exists since $l < k$) is positioned as row s in M. Because $A = M\ X = L\ T\ X = L\ A'$ and $B = M\ Y = L\ T\ Y = L\ B'$ it turns out that $\forall i = \overline{1, s-1}$, $a'_i = b'_i$, and $a'_s < b'_s$. Hence, $A \prec_s B$.

By analyzing the two cases we conclude that $\forall X \prec Y$, either $M\ X = M\ Y$ *(Case 1) or* $MX \prec MY$ *(*Case 2), hence, $M\ X \preceq M\ Y$. □

Proposition 5 *For every integral matrix M of full column rank there is a unimodular matrix β corresponding to a set of elementary column operations, such that $M\ \beta$ is an in-order matrix.*

[1] $x \prec_l y$ i.e., $x_1 = y_1, x_2 = y_2, ..., x_{l-1} = y_{l-1}, x_l < y_l$.

Proof: Let M be an $(m \times n)$ integral matrix of full column rank. Consider the $(n \times n)$ matrix T, made of the first n independent rows of M. Because M is a full column rank matrix, T is an invertible matrix (full row & column rank). By applying the HNF on T there is a unimodular matrix C corresponding to a series of elementary column operations such that $T\ C = H$, where H is in Hermite form. But *multiplication by a unimodular matrix keeps the order of the independent rows unchanged* so that, $M\ C$ is an in-order matrix as well, i.e., the first n independent rows represent H which is an in-order matrix. □

5.5.2 Multiplicity

The next proposition trivially states a necessary condition for communication without multiplicity:

Proposition 6 *If a P/C pair has a dependency function represented by a full column rank mapping matrix M then the pair is without multiplicity (M-).*

Proof: If M is full column rank then the transformation is injective such that independently of the Consumer domain $\forall X \neq Y \Rightarrow M\ X \neq M\ Y$, thus the P/C pair is without multiplicity. The Smith decomposition [67] can be used to determine whether the matrix is full column rank. □

5.5.3 An In-order example

Consider the (5×4) P/C mapping matrix:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5 & 1 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ -1 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

To decide whether two arbitrary Consumer points $(x_1, y_1, z_1, t_1)^T \prec (x_2, y_2, z_2, t_2)^T$ are mapped such that $M \cdot (x_1, y_1, z_1, t_1)^T \preceq M \cdot (x_2, y_2, z_2, t_2)^T$ we extract from top to down the independent rows of matrix M using Smith decomposition:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

and then based on Proposition 4 we conclude that the considered P/C pair is in-order.

5.5.4 An Out-of-order example

Consider the following injective P/C mapping matrix:

$$\mathbf{M} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ -5 & 1 & 0 & 1 \\ 2 & 3 & 0 & 0 \\ -1 & 6 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}.$$

Using the Smith decomposition of the matrix M to get the independent rows, we obtain the reduced mapping matrix:

$$\mathbf{T} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ -5 & 1 & 0 & 1 \\ 2 & 3 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix}.$$

According to Definition 5, T is an out-of-order transformation. Suppose the Consumer domain contains the two iterations points: $A = (0, 2, 1, 4)$ and $B = (0, 3, 1, 1)$ with $A \prec B$. When we apply to these two points the mapping M, we obtain the Producer points: $A^{'} = (0, 6, 6, 9)$ and $B^{'} = (0, 4, 9, 3)$. Because for the second entries, the relation $A^{'} \prec B^{'}$ no longer holds and the P/C pair exchanges data out of order.

5.6 Lineality space

To arrive at an accurate classification, for both order and multiplicity, based on the mapping matrix alone, there are cases for which we have to take into consideration specifics of the Consumer domain, namely whether equalities are present. Consider again the out-of-order example given in Figure 5.10. Suppose that we enclose the consumer function f_c in an if-statement as follows:

```
if ( x == 1 ) {
   fc(a[y,x])
}
```

In this case, the mapping matrix M remains the anti-diagonal $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ and in the context of Proposition 4, we may think that the corresponding P/C pair is out-of-order. But, since the Consumer domain has now an equality (i.e. $x = 1$), this P/C pair becomes actually an in-order P/C pair although the corresponding mapping matrix is out-of-order, as shown in Figure 5.11.

In general, equalities in the Consumer cause problems in the correct classification of order and multiplicity. To avoid the kind of 'miss' classification as shown in Figure 5.11, we propose in this section a solution that consists of deriving a new in-order matrix C_{new} that maps a consumer domain containing equalities into a new consumer domain without equalities and without changing the order and multiplicity properties. This new in-order matrix is obtain using the Hermite Normal Form (HNF). In the next section, we first explain the new procedure and then give two examples. In the first example, we look at order and in the second example we look at multiplicity.

For the case given in Figure 5.11, based on the new procedure, it turns out that the first column of M is irrelevant due to the equality and instead we need to consider the matrix $M^{'} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$. This matrix does satisfy Proposition 4, and the PC is classified correctly as in-order without multiplicity.

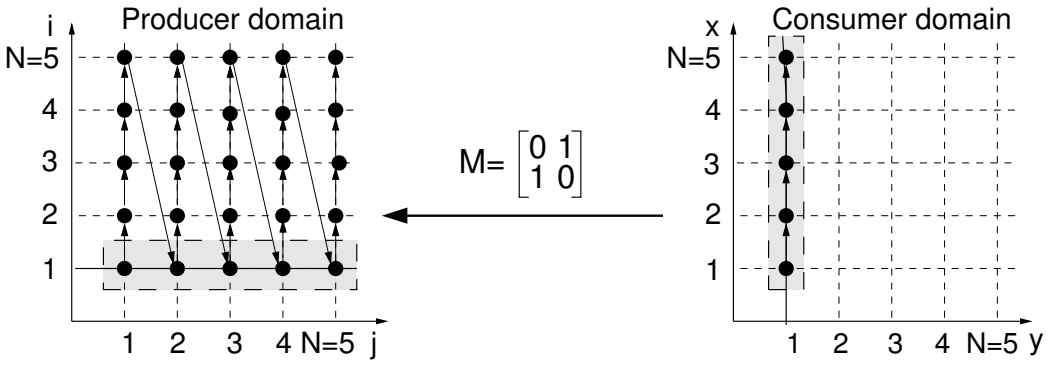


Figure 5.11: An in-order P/C pair characterized by an out-of-order mapping matrix

5.6.1 Ordering and multiplicity characteristics in the presence of a Consumer lineality space

Consider a Consumer domain that is described by a set of constraints containing a number of equalities, i.e., the domain has a *lineality space*. This space is described by the matrix L that contains the equalities. In this situation, we propose a procedure that derives an in-order matrix C_{New} that maps (in-order) a fully dimensional domain D' into the Consumer domain D. Matrix C_{New} is derived starting from the matrix C given by the HNF of L. Once we have C_{New}, we show that if MC_{New} (See Figure 5.12), is an in-order transformation, then the initial P/C pair is also in-order. In Lemma 1, we show how C_{New} can be obtained from an arbitrary matrix C, as given by the HNF of L. Therefore, instead of analyzing only the P/C mapping matrix M, we also have to take into consideration the equalities (lineality space) that describe the Consumer domain. This is graphically depicted in Figure 5.12.

Proposition 7 *Consider an n-dimensional polytope $\mathcal{D}$ and a mapping function f represented by a $(n \times n)$ unimodular matrix C. Consider the linearly bounded lattice:*

$$P(p) = f(\mathcal{D}(p) \cap Z^n) = \{i \in Z^n \mid i = Ck + O \wedge \ k \in (\mathcal{D}(p) \cap Z^n)\}$$

and the polytope:

$$P'(p) = f(\mathcal{D}(p)) = \{x \in Q^n \mid x = Ck + O \wedge \ y \in (\mathcal{D}(p) \cap Q^n)\}.$$

Hence, the next two sets are equal:

$$P(p) = P'(p) \cap Z^n.$$

Basically, the previous property states that the image of a polytope through a unimodular matrix does not introduce holes.

Lemma 1 *For any n-dimensional polytope domain $\mathcal{D}$ that has a $(n-p)$ dimensional lineality space given by a $(p \times n)$ matrix L, there always exists a n-dimensional polytope domain $\mathcal{D}'$ with the leading p coordinates constants, and a unimodular matrix C_{New} such that, $\mathcal{D}'$ is mapped in-order by C_{New} into $\mathcal{D}$.*

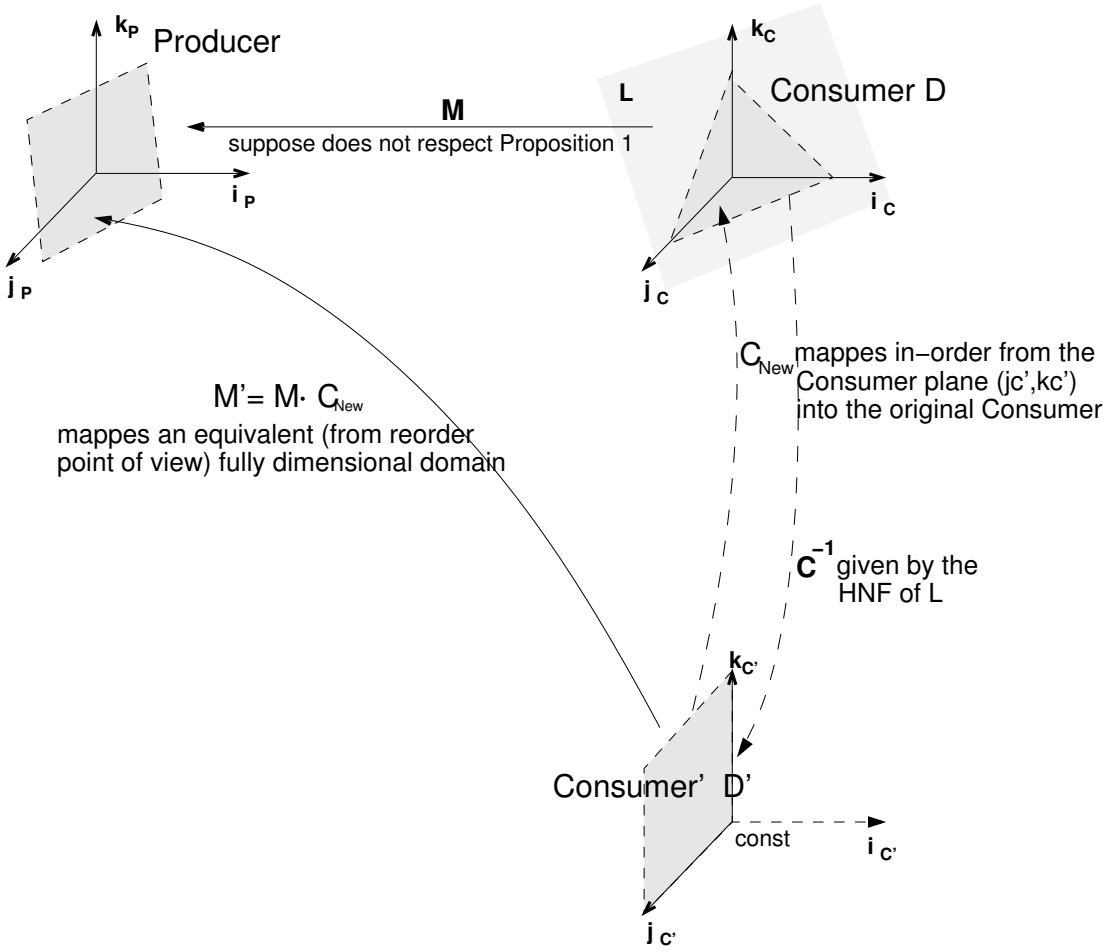


Figure 5.12: Out-of-order detection schema, when the Consumer domain has a lineality space

Proof: Let the lineality space of the domain $\mathcal{D}$ be represented by the full row rank matrix L (otherwise $\mathcal{D}$ is empty). Applying Hermite decomposition L is expressed as $L\ C = [H, 0]$, where H is a $(p \times p)$ Hermite matrix and C is an $(n \times n)$ unimodular matrix. Let $C = [C_1, C_2]$, where C_1 consists of the first p columns of C and C_2 of the last $n - p$ columns of C. Hence,

$$L\ C_1 = H \wedge L\ C_2 = 0_{n-p}. \tag{5.6}$$

From Proposition 5, there exists a unimodular matrix β such that $C_2' = C_2\ \beta$ is an in-order matrix. But multiplication by β represents a series of elementary operations on the last $(n - p)$ columns of C. Hence, the matrix $C_{New} = [C_1, C_2']$ is unimodular, with C_2' an in-order matrix such that $L\ C_{New} = [H, 0]$.

Consider now $\mathcal{D}'$ as the integral image of the polytope $\mathcal{D}$ through the matrix C_{New}^{-1}. But C_{New}^{-1} is integral unimodular as the inverse of an integral unimodular matrix. Applying Proposition 7, $\mathcal{D}'$ is a polytope. We now show that $\mathcal{D}'$ is mapped in-order by C_{New} into $\mathcal{D}$. Let the equalities in the Consumer domain be described as:

$$L\ X_C = q, \tag{5.7}$$

where X_C is an arbitrary point in $\mathcal{D}$ and q is a constant vector. Substituting $L = [H, 0]\ C_{New}^{-1}$ in equation 5.7 we get:

$$[H, 0]\ C_{New}^{-1} X_C = q.$$

Because $C_{New}^{-1} X_C = X'_C$ we get that:

$$[H, 0]\ X'_C = q,$$

where $X'_C \in \mathcal{D}'$. Because H is bijective, $X'_C = \begin{bmatrix} \alpha_1 \\ : \\ \alpha_p \\ x'_{p+1} \\ : \\ x'_n \end{bmatrix}$, where the first p elements are constants given by $\begin{bmatrix} \alpha_1 \\ : \\ \alpha_p \end{bmatrix} = H^{-1}q$ and the last $(n-p)$ elements x'_i are free variables. Because the first p elements of X'_C are constants, only the last $(n-p)$ columns of C_{New} play a role from the reordering point of view when mapping from domain D' to domain D. But the last $(n-p)$ columns of C_{New} represent C'_2 which is an in-order matrix. Therefore, C_{New} maps in-order a fully dimensional domain D' into.the original consumer domain D. □

Below, we generalize Proposition 4 by giving a necessary condition for in-order communication in the case when an n-dimensional Producer domain contains a p-dimensional lineality space:

Theorem 4 *If the last $(n-p)$ columns of the matrix $M\ C_{New}$ form an in-order injective mapping, then the P/C characterized by M is an in-order pair.*

Proof: Because C_{New} is unimodular the inverse C_{New}^{-1} exists and is integral. Consider two arbitrary Consumer points A, B with $A \prec B$. They will be mapped through the matrix C_{New}^{-1} to the Consumer' points A' and B' with $A' \prec B'$. Applying Lemma 1, $M\ C_{New}$ maps A' and B' to two Producer points A'' respectively B'' with $A'' \prec B''$. But A'' and B'' are also the Producer images of A and B through the mapping M, hence the P/C pair is in-order. □

A necessary condition for avoiding multiplicity is that the mapping matrix should represent an injective transformation. Otherwise, multiplicity may be involved, i.e., a produced token will be consumed by more than one Consumer iterations. As shown in the example in Section 5.7, this condition is not necessary, as there are particular cases where the mapping is not injective yet multiplicity does not appear. Again this arrives when the Consumer iteration space has a lineality space. If so, we make use of the procedure for removing the lineality space as described in the previous section, obtaining a necessary condition for communication without multiplicity:

Corollary 1 *If $M\ C_{New}$ restricted to the new Consumer domain C', represents an injective mapping, then the corresponding P/C pair is without multiplicity.* □

In the next example, we illustrate, based on Lemma 1 and Theorem 4, the procedure for deciding on the P/C communication type if the Consumer domain lies on a hyperplane in a higher dimensional space:

5.6.2 Example - Reordering in case of a Consumer lineality space

Consider the 3-dimensional Consumer domain X that lies on the plane given by $j_C - k_C = 0$ as shown in Figure 5.13. This plane is described by the matrix: $L = [\,0\ 1\text{-}1\,]$. Suppose the mapping matrix is $M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

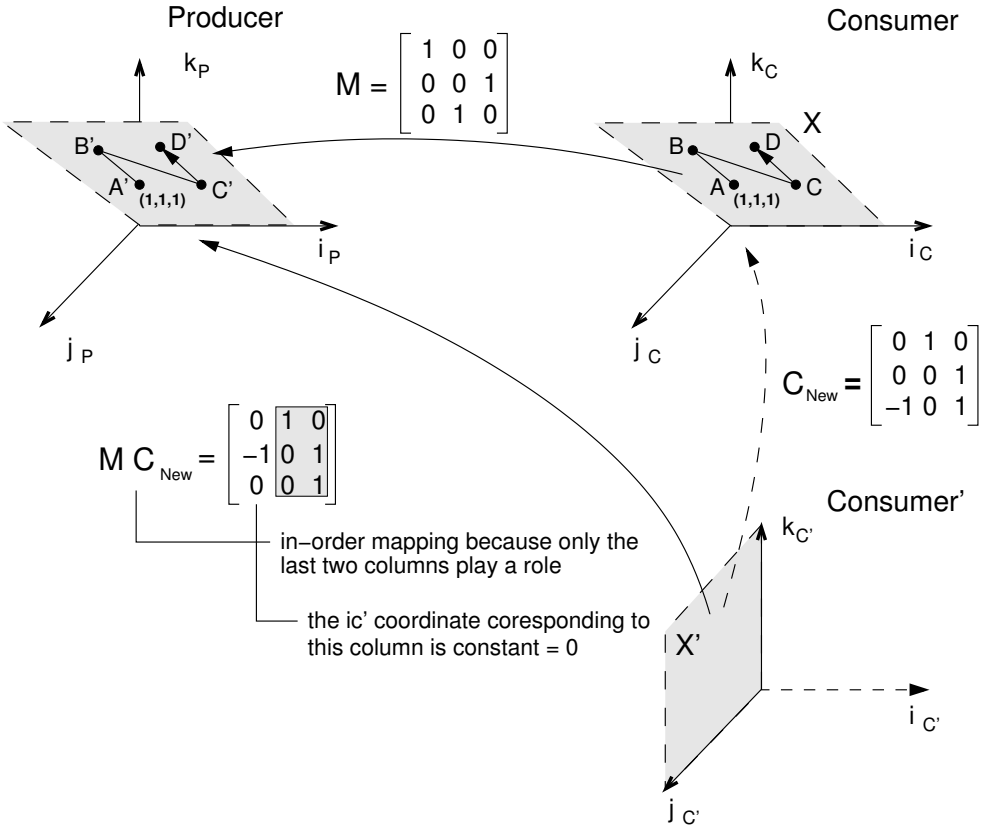


Figure 5.13: An in-order example

According to Proposition 4, the Producer and the Consumer may not form an in-order pair. However, as can be observed in the top part of Figure 5.13, the Consumer points $A(1,1,1) \prec B(1,2,2) \prec C(2,1,1) \prec D(2,2,2)$ are mapped in-order through M into the Producer points $A'(1,1,1) \prec B'(1,2,2) \prec C'(2,1,1) \prec D'(2,2,2)$. Because the Consumer domain includes a lineality space, we apply Theorem 4. Here we show how the matrix C_{New} is constructed. Using the HNF of L, the corresponding unimodular matrix $C = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ is computed, such that $[H, 0] = L\ C^{-1} = [1\ 0\ 0]$. According to matrix C, this domain will be mapped (see Figure 5.13) into a domain of the form $X' = \begin{bmatrix} i_{C'} = 0 \\ j_{C'} \\ k_{C'} \end{bmatrix}$. Because the first coordinate in the X' domain is constant

(equal to 0), only the last 2 columns of C play a role from the reordering point of view when mapping from X' to X. According to Proposition 4, C does not map in-order from X' to X, so C is not the matrix we are looking for. However, we make use of Lemma 1 that provides the column transformation on C that lead to the in-order matrix C_{New} which keeps the initial Hermite form of L.

Hence, we take out from C the 2^{nd} and the 3^{rd} column (since the first column corresponds to a constant coordinate). The new matrix becomes:

$$C_2 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}.$$

We take out the first two independent rows from C_2, obtaining the matrix:

$$C_{Indep} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Because C_{Indep} is a full column & row rank matrix, it can be converted to an in-order Hermite matrix by multiplication with the matrix: $\beta = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.

We multiply C_2 by β and we get the in-order matrix: $C_2' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$. Now we attach to the matrix C_2' the first column of C and we get the matrix C_{New}, that is in fact obtained from C by performing elementary operations on the columns corresponding to the zero columns from the HNF of the matrix L. Therefore, by multiplying L by C_{New} we get the same Hermite normal form: $[H, 0] = L\ C_{New} = [\,0\ 1\text{-}1\,] \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} = [\,1\ 0\ 0\,]$. The difference is that now the matrix C_{New} is an in-order mapping. Now we make the product between M and C_{New}: $M\ C_{New} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$. From this new matrix we remove the first column (which corresponds to the constant coordinate $i_{C'}$) obtaining the matrix: $T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$, which is an in-order matrix. Due to Theorem 4, we conclude that the P/C pair is in-order.

5.6.3 Example - Multiplicity

Consider the P/C pair shown in Figure 5.14, where the Consumer domain is $\{(i,j) \in Z^2 \mid 1 \leq i \leq 6, i = j\}$. The dependency matrix is $M = [1\ 0]$. It represents a non-injective mapping, yet the pair is actually without multiplicity. Using Corollary 1, we detect automatically that the P/C pair is without multiplicity. Following the same procedure as in Section 5.6.1, we first obtain $C_{New} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, and the product: $M\ C_{New} = \begin{bmatrix} 1 & 1 \end{bmatrix}$. We remove the first column of this matrix (which corresponds to the constant coordinate $i_{C'} = 0$) obtaining matrix $T = \begin{bmatrix} 1 \end{bmatrix}$, which represents the injective transformation $f(j) = j$. From Corollary 1, we conclude that the P/C pair is without multiplicity.

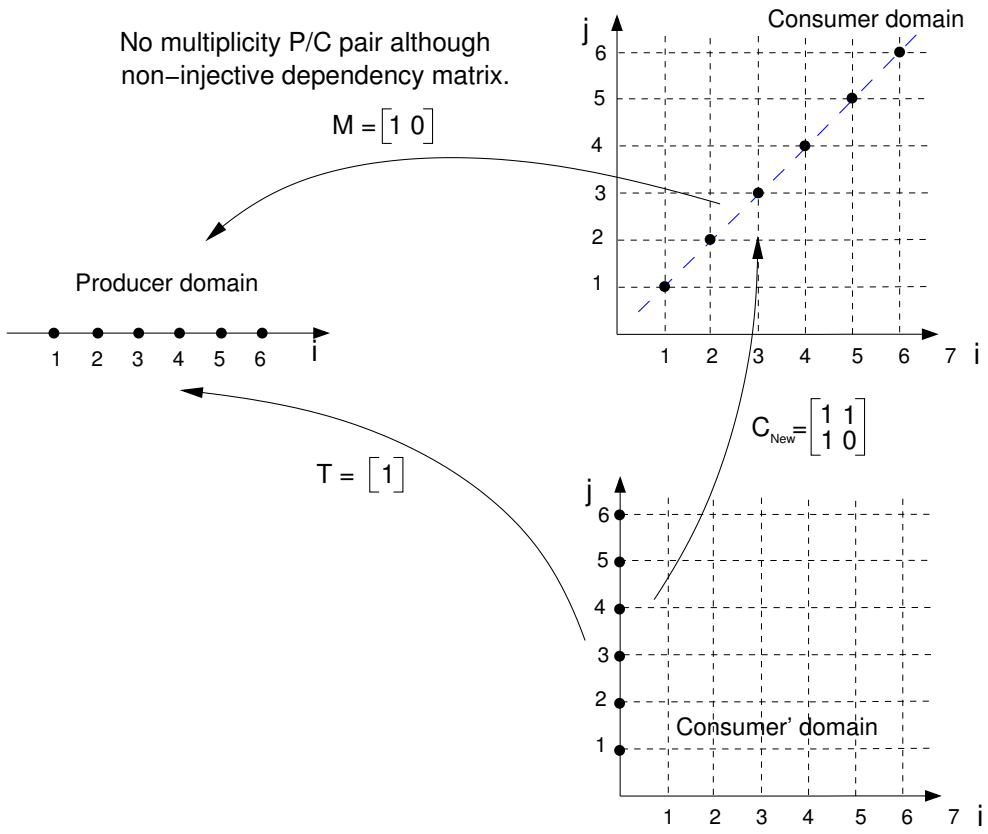


Figure 5.14: Multiplicity example with no full column rank mapping matrix

5.7 Limitations

Although we can classify most of the P/C pairs using the propositions presented in Section 5.5 and Section 5.6, there are still cases that cannot be classified correctly using those propositions. In such cases, we require the use of the ILP formulation for MT and RT. In Figure 5.15, we show two cases for which we need to rely on RT and MT to correctly classify the communication types. In the figure, two P/C pairs are graphically depicted. Both pairs have the same Consumer IS: $\{(i,j) \in Z^2 \mid 2 \leq i \leq 10, 2 \leq j \leq N, i+2 \leq 3*j \leq i+4\}$. This Consumer IS does not contain equalities.

The top P/C pair has a non-full column rank dependency matrix $M = [\,1\ 0\,]$. Based on Proposition 6, and because the Consumer has no lineality space, we would conclude that multiplicity is involved. However, if we analyze the P/C pair in Figure 5.15, we see that this P/C pair is actually without multiplicity. To get an accurate decision in this case, we need to use the MT.

The bottom P/C pair has the dependency matrix $M = [\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}]$. Based on Proposition 4, we would conclude that the P/C pair is out-of-order. However, if we analyze the P/C pair in Figure 5.15, we see that this P/C pair is actually in-order. To get an accurate decision in this case, we need to use the RT.

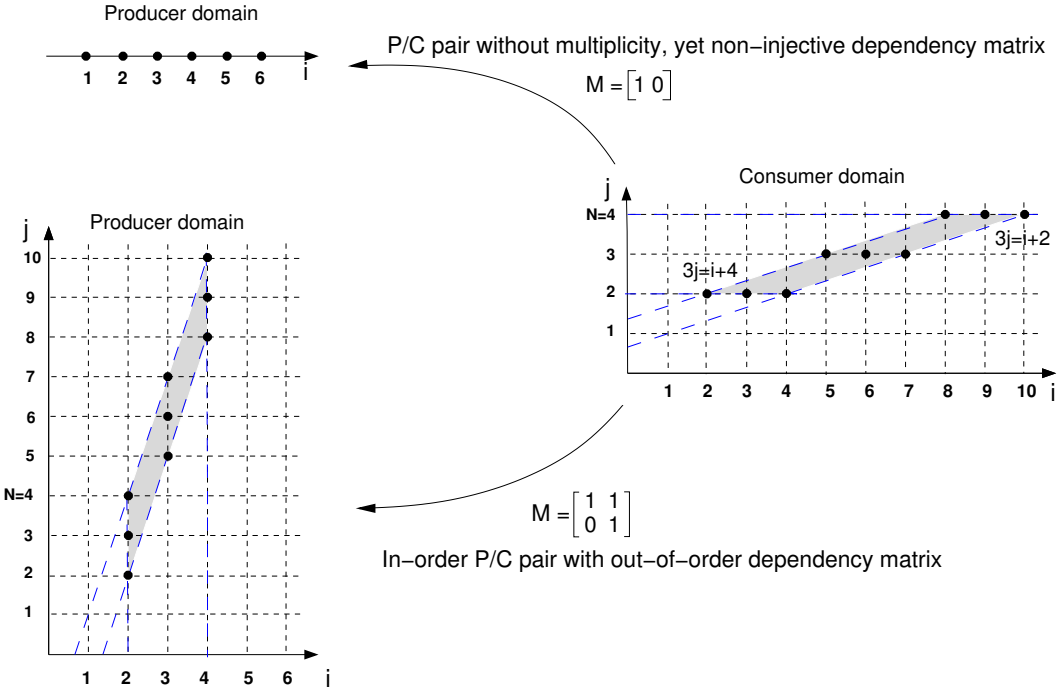


Figure 5.15: The classification requires the ILP formulation of MT and RT tests

5.8 Putting things together - a hybrid approach

Although we can classify most of the P/C pairs using the propositions presented in Section 5.5 and Section 5.6, there are pairs that cannot be correctly classified using these propositions. In such cases, we do require the use of the ILP formulation for the MT and RT, as shown in Section 5.7. Combining the presented techniques so far, leads to a hybrid Reordering/Multiplicity decision tree as shown in Figure 5.16. This decision tree classifies each P/C pair, but uses less times integer linear programming then when using only the RT and MT.

Using this decision tree, we see how each P/C pair is correctly placed in one of the four communication types (IOM-, OOM-, IOM+, or OOM+) given at the bottom of the decision tree. First we check whether the Consumer C has lineality space. If so then Theorem 4 or Corollary 1 is applied, and the Consumer domain is transformed into a fully dimensional domain without introducing reordering or multiplicity. After this pre-processing step, we determine whether the dependency matrix M is of full column rank (Proposition 6), using Smith decomposition. When the test determines that M is not full column rank, although a possible lineality space has been removed, it might still be that multiplicity is not involved as shown in Section 5.7, and we use the Multiplicity Test to be sure. The other way around, when the matrix is full rank, the P/C pair is guaranteed without multiplicity. Next, we apply Proposition 4 to check whether the dependency matrix is in-order. If yes, then we are sure that the P/C pair is in-order. Otherwise, it might still be that reordering is not involved as shown in Section 5.7 and we use the Reordering Test to be sure. In this way we classify each P/C pair to one of the four possible types.

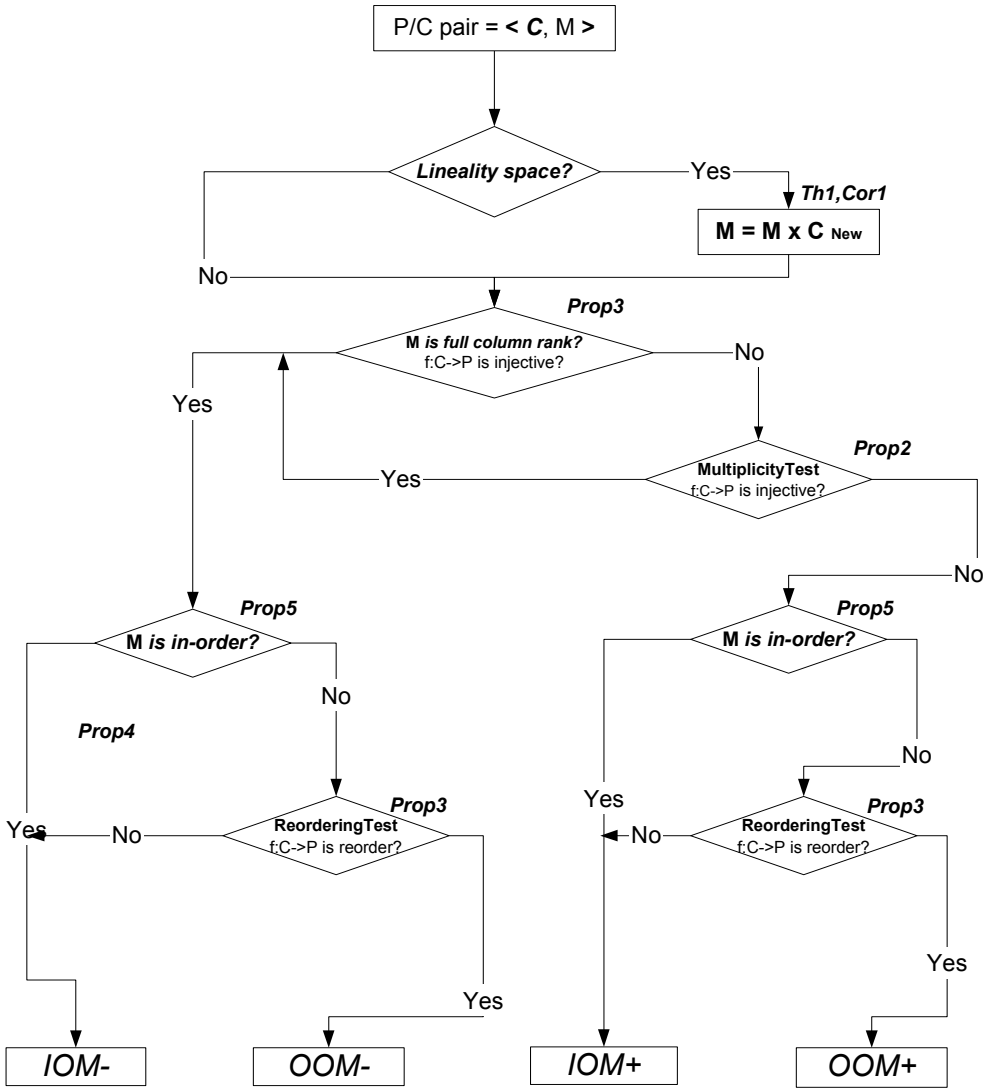


Figure 5.16: The Hybrid Reordering/Multiplicity Decision Tree

5.9 Results

The Hybrid Reordering/Multiplicity Decision Tree shown in Figure 5.16 has been implemented in Compaan, realizing the Communication Model Selection step. In Table 5.2, we present again the eight network configurations for image and signal processing algorithms as we did in Table 5.1. In the table, we show how many times a Reordering Test (RT) and Multiplicity Test (MT) is done in the hybrid classification scheme. The values given between brackets are the number of tests performed when only the RT and MT are used. The classification that uses only MT and RT, solved 4516 systems for the Reordering test and 1780 systems for the Multiplicity test for the 8 examples. Using the decision tree presented in Figure 5.16, only 237 systems are solved for the Reordering test and 102 systems for the Multiplicity test. This means that the hybrid classification scheme reduces the number of system to be solved by 95%. In only 5% of the cases is integer linear programming used. In

95% of the cases, the properties presented in Section 5.5 and Section 5.6 are able to correctly classify a P/C pair, leading therefore to a big reduction in compute time and memory requirements.

Algorithm	P/C pairs	RT		MT	
QR-Decomp	12	0	(73)	0	(30)
SVD	118	171	(1283)	51	(565)
Faddeev	28	4	(205)	7	(78)
Gauss-Elimin	11	17	(17)	6	(6)
BeamFormer	98	18	(408)	4	(196)
Motion Estim	98	0	(882)	0	(294)
M-JPEG	50	24	(178)	34	(93)
Stereo Vision	173	3	(1470)	0	(518)

Table 5.2: Classifying P/C pairs in real-life examples to the four communication types using the Decision Tree.

5.10 Discussion and conclusions

In this chapter, we presented the problem of classifying the communication types of P/C pairs of statements in a nested for-loop program. The classification distinguishes between four patterns of communicating data and is needed to break down a sequential program into autonomously running processes.

To classify a P/C pair to one of the four types, we presented in the first part of the chapter two tests. The Reordering test determines whether re-order occurs in the communication between an arbitrary P/C pair. The Multiplicity test determines if reuse of data takes place on the communication between an arbitrary P/C pair. Both tests have been formulated as integer linear programming problems. Both tests are memory intensive and time consuming, and therefore, we presented in the second part of the chapter additional tests that are polynomial time algorithms. By combining these additional tests with the original Multiplicity and Reordering tests, we derived a hybrid classification scheme that exactly classifies all interprocess communication types. Using the hybrid classification scheme, we use in only 5% of the cases integer linear programming; in the remaining 95% of the cases, we use polynomial time algorithms. This makes the hybrid approach well suited for implementation in a compiler like the Compaan Compiler. We have tested and classified eight real-life examples to validate our approach.

Chapter 6

Process Network Synthesis

In the first two compilation steps, i.e., Production and Consumption Restructuring, we have presented how the storage structure of an application can be restructured such that each P/C pair of assignment statements communicate a data argument over a dedicated memory array. From now on, when referring to a **sequential program**, we mean the code obtained after performing those two compilation steps. Furthermore, with the analysis done during the third compilation step, we are able to characterize the communication type of each P/C pair of assignment statements. The purpose of these three steps is to support the conversion of sequential programs to concurrent programs, i.e., from sequential imperative specification to a parallel language specification, in particular process networks (PNs). The PN model of computation used as a target by our compiler resembles the *Kahn Process Network (KPN)* model of computation [51]. Between our PN and KPN there are two major differences: **1.** the PN channel sizes are bounded without introducing network deadlock and **2.** the PN communication is not restricted to always take place in a FIFO manner. The PN communication channel uses *random addressable memory (RAM)* or *content addressable memory (CAM)*, which allow the order of consuming data to be different from the order of producing it. Consequently, a FIFO channel is just a particular representation of a RAM/CAM channel. Nevertheless, because of the streaming characteristics of the class of applications that we consider for compilation, the resulted networks will in most of the cases communicate data over FIFO channels. This implies that as a result of our compilation, the network will most likely be a KPN. To conclude, our PN model of computation has the following characteristics:

- each process is a sequential program that consumes tokens from its input channels and produces tokens to its output channels,
- each communication channel has one source and one destination and can be accessed randomly,
- a process is blocked when it tries to read from an empty channel,
- although in Kahn Process Networks writing is non-blocking because the channels have unbounded capacity, our PN model, uses bounded communication channels, and hence, needs blocking writes as well.

6.1 Problem Definition

In this chapter, we address the problem of converting a sequential program to a parallel representation. For this purpose, our compiler derives a PN according to the basic description given in the beginning of this chapter. Obviously, the PN is input/output equivalent to the sequential program, and a **firing** of a process corresponds to the execution of one of the assignment statements at a scheduled step in the sequential program execution. Each process of the PN makes progress following a local schedule, without following a global schedule over the whole network. In this way, at a particular moment during the execution of the network, several processes may be active simultaneously. This situation corresponds to a parallel execution of the assignment statements in the sequential program.

6.2 Solution

To arrive at a PN, we first convert the sequential program to an intermediary process network (IPN), where all the executions of one assignment statement are collapsed into a single *process* and the memory arrays are becoming communication *channels*. Basically the IPN provides a restructured representation of the sequential program. Recall that after the first two compilation steps, each producer/consumer pair of assignment statements communicate a data argument over a dedicated memory array. Due to the polyhedral analysis done so far, the IPN structure is already available from the previous compilation steps. As shown in Figure 6.1, the IPN processes are represented by circles connected by edges representing unidirectional communication channels. The topology of this network resembles the Reduced Data Dependence Graph [26] of the sequential program from which it originates. The communication channels are random access memories, so before reading/writing a consumer/producer process has to compute a read/write address. Next to the IPN presented in

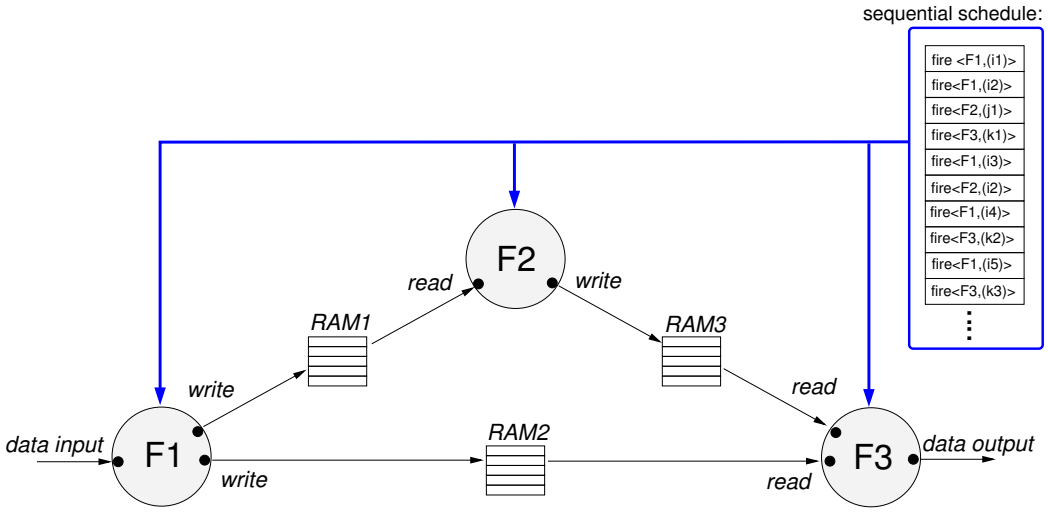


Figure 6.1: An example of an Intermediary Process Network together with the sequential schedule of firing the component processes

Figure 6.1 is shown, a sequential schedule that fires the processes one at a time following the global

schedule according to which the assignment statements are executed in the sequential program. An IPN is characterized by:

- A collection of processes where an arbitrary process F_i is characterized by:
  - a function call F_i corresponding to an assignment statement in the sequential program
  - a number of input ports as provided after performing the Consumption Restructuring step. Each input port represents a set IP of iteration points and corresponds to one of the process incoming channels. Depending on the iteration point associated to a firing, at each process firing some of the input ports are active, i.e., the data is read from their incoming channels. To each input port corresponds a *read function*, $read : IP \rightarrow Z^n$ that maps the producer iteration space into the memory space, i.e., for each iteration point the read function outputs the address from where the process has to read a token.
  - a number of output ports as provided after performing the Production Restructuring step. Each output port represents a set OP of iteration points corresponding to one of the process outgoing channels. At each process firing, some of the output ports are active, i.e., the data is written to their outgoing channels. To each output port corresponds a *write function*, $write : OP \rightarrow Z^n$ that maps the consumer iteration space into the memory space, i.e., for each iteration point the write function outputs the address where the process has to store a token.
- A communication structure consisting of unidirectional channels identical to the memory structure of the sequential program. Each channel connects a producer output port OP to a consumer input port IP. To each channel corresponds a data dependency function $f : IP \rightarrow OP$ which relates the iterations of the IP to the iterations of the OP. As we show in Section 6.2.1, for each channel there exist as well a function (called lexicographically minimal/maximal preimage) that relates the iterations from OP to the first/last consumptions from IP: $lmp/lMp : IP \rightarrow OP$.
- A global schedule determining the order in which the processes of the IPN have to fire. This schedule is given by the order in which the assignment statements corresponding to the processes are executed in the sequential program. The schedule contains tuples $< F_i, ip >$, where F_i represents the name of the process to be fired and ip is the iteration at which the corresponding assignment statement is fired in the sequential program. For example $< F_i, j >$ represents the firing of the process F_i at iteration point j and corresponds to the execution of assignment F_i at iteration j in the sequential program.

There are a few important properties of an IPN. First, the schedule of an IPN guarantees that each process fires over the entire iteration space, and second, the network uses the same amount of memory and is input/output equivalent to the sequential program.

Running example

The IPN for our running example is graphically depicted in Figure 6.2. The execution of each assignment statement over its iteration space is collapsed into a process with a number of input and output ports. As can be seen at the right side in the picture, the sequential schedule of firing the processes in the IPN is given as a nested loop program executing at different iteration points a **fire** actions,

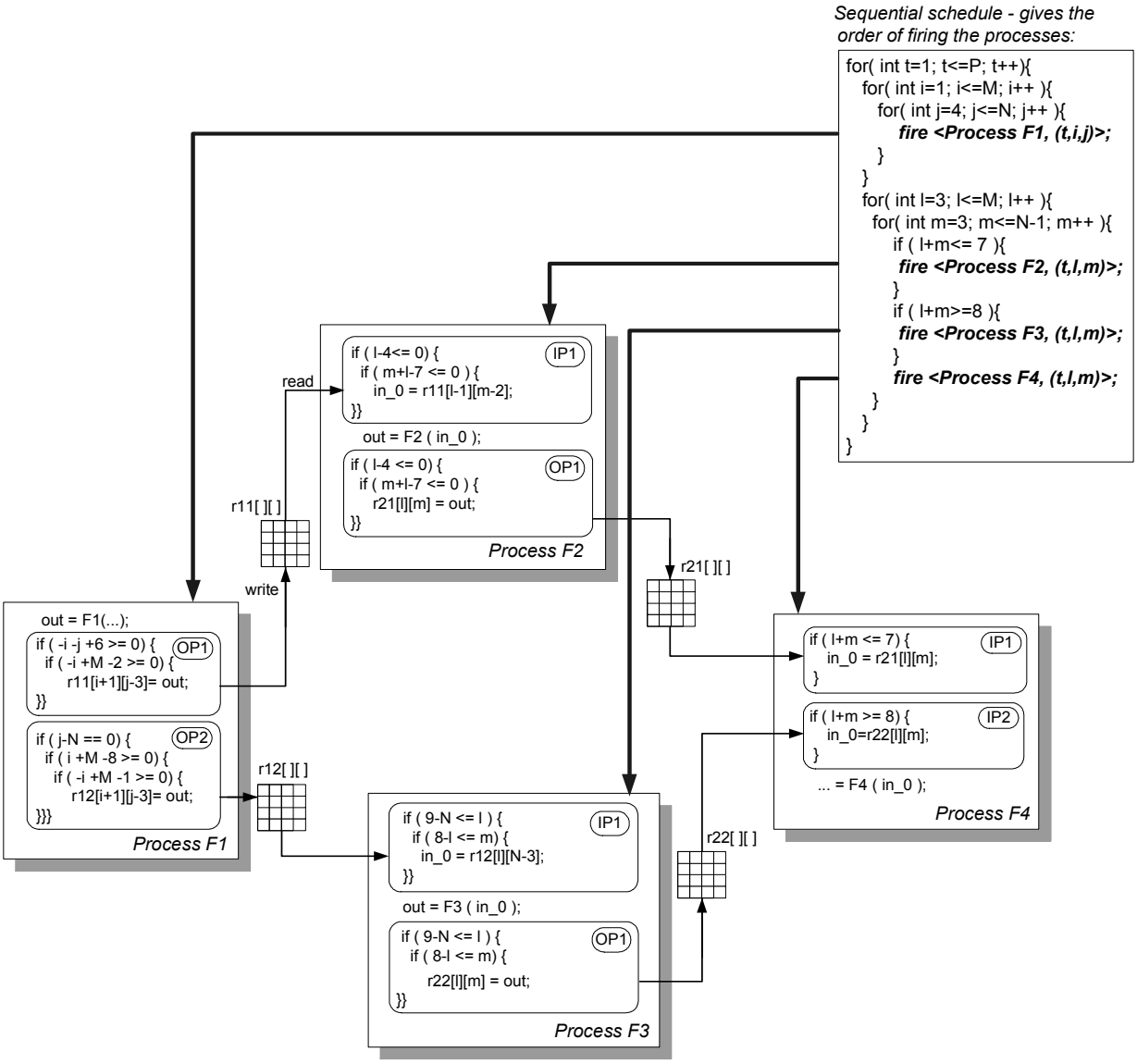


Figure 6.2: Intermediary Process Network together with a sequential schedule

e.g., $fire < Process\ F1, (t,i,j) >$,$fire < Process\ F2, (t,l,m) >$, $fire < Process\ F3, (t,l,m) >$ and $fire < Process\ F4, (t,l,m) >$. This sequential schedule follows the order in which the corresponding assignment statements are executed in the sequential code, and activates the processes by passing them an iteration point. For example the iteration point (t,i,j) is passed to process $F1$. Depending on the iteration point that is passed to the process, some of the input channels are read and some of the output channels are written to. For example, when firing process $F2$ at iteration point $(t=1, l=3, m=3)$, port $IP1$ is activated, and data is consumed from $r11[2][1]$, as the read function is $read(t,l,m) = (l-1, m-2)$. Then the data is passed as input to the internal function $F2$ of which output as a result of activating $OP1$ is written at location $r21[3][3]$ based on the write function $write(t,l,m) = (l,m)$.

Notice that the sequential schedule shown in Figure 6.2 is taken from the original code. The reason is that the order of executing assignment statements in the code obtained after Production and Consumption Restructuring is the same as in the original code.

6.2.1 Removing the global schedule

Now that the intermediary process network has been introduced, the following question arises: *is it possible to remove the sequential global schedule and to allow overlapping the execution of the processes?*

In this section, we present how a process network can be synthesized from an IPN by replacing the global schedule with what we call an adequate synchronization policy. This synchronization policy allows independent and concurrent execution of the component processes. Figure 6.3 depicts the topology of the asynchronous process network (PN) obtained from the one give in Figure 6.1. A process part of a PN follows a local schedule that respects the same order as the corresponding process is fired in the IPN; the part of the global schedule representing the firings of one and the same process has been embedded internally to the process part of the PN. We thus, have the following property:

Property 1 *The firings of a process part of a PN follow the same order as the firings of the same process in the IPN as well as the order in which the correspondent assignment statement is fired within the sequential program.*

The processes in a PN are communicating data point-to-point over unidirectional storage channels. The storage is random access memory that is accessed with the same allocation scheme (i.e., read and write functions) as in the IPN. To ensure that the PN is functionally equivalent to the IPN from which originates, a synchronization policy among the communicating processes has to be derived. According to this policy, when accessing a memory location, the producer/consumer process has to perform a certain *blocking-write/blocking-read* primitive.

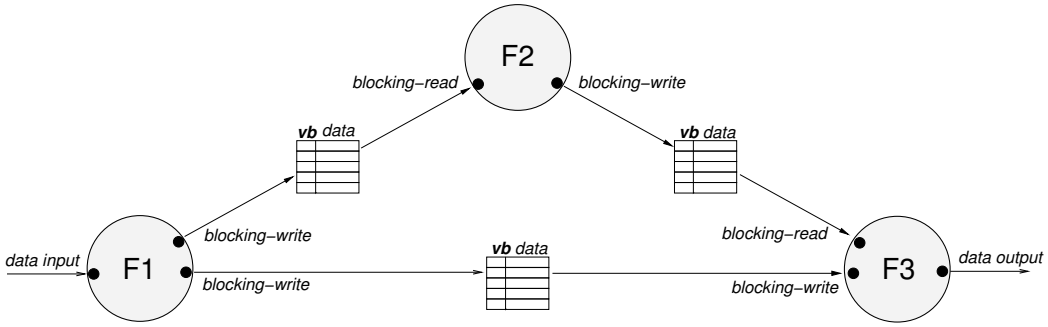


Figure 6.3: Process Network with an inter process synchronization with blocking-read and blocking-write

Synchronization policy

According to Berstein's conditions [10], formally addressed by Ken Kennedy and Randy Allen in their *Theorem of Dependence* [56], two statements of a sequential program can be executed in parallel (or independently of each other) without violating the correctness of the program if no true dependence exists between them i.e., no flow, anti or output dependency[1] exist.

Therefore, parallel execution with the same functionality as the sequential program becomes possible by respecting the flow, anti and output dependencies. In the IPN (as in the program derived after the first two compilation steps) only two types of dependencies are present:

- output dependencies - between firings of the same process relative to the same channel location, and
- flow dependencies - between producer/consumer firings [2].

The output dependencies between several iterations of a single statement are respected due to the internal process schedule, i.e., internally the process follows the same schedule in which the corresponding statement was iterated within the sequential code. Therefore, no output conflicts can appear. The flow dependencies are respected, by imposing an adequate inter-process synchronization policy. This is done by an-notating each location a memory array used to communicate data with a boolean called *validity bit* (vb). Thus, when writing to a memory address where $vb = 0$, the producer can commit the write (or overwrite) action at that location. After a write action, the producer sets the validity bit $vb = 1$. If a consumer process attempts to read data from a memory location where $vb = 0$, then the reading action is stalled until the producer writes at that location and sets $vb = 1$. When the consumer finishes consuming data from an address, it sets the validity bit to $vb = 0$ indicating that the producer process can overwrite that memory location. This situation corresponds to the case where $write : OP \rightarrow Z^n$ is not an injective function. Observe that a token can be consumed more than once by the consumer process, such that only after the last consumption has taken place, the consumer sets $vb = 0$ allowing the producer to overwrite. This corresponds to the case where $read : IP \rightarrow Z^n$ is not an injective function and multiplicity is involved. Hence, each Consumer has to be able to determine by itself (based on its ports description) what is the last consumption of a token from a certain memory location[3]. As shown later in Section 6.2.1, the behavior of the validity bit will be synthesized based on a compile time procedure.

The synchronization based on the validity bit guarantees that the PN is input/output equivalent to the sequential program. Thus, for example the PN will not deadlock. For a deeper insight regarding the transformation, specially from the point of view of deadlock we present the following theorem.

Theorem 5 - the PN deadlock theorem: *A PN is guaranteed to be free of deadlock and is input/output equivalent to the IPN from which it is derived.*

Proof: First we prove that for the same data input as provided to the original program, within the PN all the processes finish their execution. Suppose this is not the case such that at a moment all

[1] Observe that this concept of dependency differs from the concept of dependency in data-flow analysis [1, 66] in which a statement has to be executed before or after another in order to receive the correct input data.

[2] A single process can be both consumer and producer, in which case the communication takes place over an intraprocess communication channel.

[3] Recall that the sequential program is such that each token that is produced will be consumed.

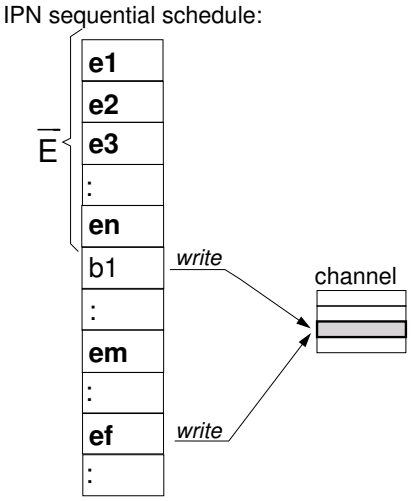


Figure 6.4: Filling the parallel PN execution into the corresponding IPN schedule

processes in a PN are blocked (on reading or writing data) but there are still processes to be further fired. Let $E = \{e_i\}$ be the set of already executed process firings and $B = \{b_j\}$ the set of blocked firings. Let $\overline{E} = \{e'_t \in E\}$ be the subset of E that recovers a continuous (sequential) sequence of firings within the sequential IPN schedule. For example in Figure 6.4 this set is represented by $\overline{E} = \{e1, e2, ...en\}$, while E additionally may include other firings, e.g, (em).

According to our supposition B is not empty. Let $b1$ be the element of B that follows in the sequential schedule the last scheduled element of $\overline{E}$. For the execution of $b1$ the required data has been produced (the sequential schedule is valid) and this data is still available (guaranteed by the synchronization policy in which a memory location in a communication channel is overwritten only after the last consumption has taken place). Thus, the process corresponding to the firing $b1$ is blocked on writing data in a channel. This means that there is another firing ef that has written in the same channel at the same location, and that data is still to be used. The fact that only at firings of the same process data is written in the same channel, implies that ef represents a firing of the same process as $b1$. However, the sequential IPN schedule guarantees that $ef \notin \overline{E}$; the precedence of firings in the IPN schedule guarantees no writing conflicts - the last consumption from an address took place before an overwriting at that address is performed. This implies that while in the parallel schedule ef is scheduled before $b1$, in the sequential code their order is reversed. This conclusion contradicts Property 1 stating that the internal process schedule respects the sequential order in which the correspondent assignments were fired in the schedule of the sequential program. Hence, the made supposition is false so all processes finished their execution.

Secondly, at an arbitrary iteration i process F_m consumes data produced at iteration j by process F_n (i.e., $F_m^i \delta F_n^j$) if and only if in the sequential program the assignment F_m at iteration i consumes data produced by assignment F_n at iteration j. By means of induction it turns out that if in the sequential program the function F_m at iteration i inputs one of its arguments $in_k = F_{l1} \circ ... \circ F_{ls-1} \circ F_{ls}(it_1)$ where it_1 is an input token, then in the PN the process F_m at firing i consumes from one of its channels $in_k = F_{l1} \circ ... \circ F_{ls-1} \circ F_{ls}(it_1)$. □

6.3 Converting IPN to PN

In this section we describe the conversion of the IPN to an input/output equivalent PN representation. First we show how the processes with their local schedules are created. Then how the synchronization policy is derived and finally the synthesis of the inter-process communication structure.

6.3.1 Process local schedule

A PN process firing sequence follows the sequence of execution of the corresponding assignment statement in the sequential program or IPN. Therefore, one possibility to get the local schedule is to enforce internally to each process a loop nest structure that (iterates) over the same iteration space (IS) as the corresponding statement in the sequential program. The for-loops used for iterating over the IS of an assignment statement will be the same as the ones used for firing a process. Due to the intermediate polyhedral representation of the loops in the sequential program, extra optimizations (dealing with the reduction of the control) based on Fourier-Motzkin Elimination [22] or the Omega test [73] can be achieved [77, 55, 23]. Further on, the IPDs and the OPDs derived in the first two compilation steps, are transformed into sets of if statements with affine expressions. Certain optimization like removal of the redundant if-statements can be obtained using PolyLib or Omega. Those components (i.e., processes, process iteration spaces, IPDs and OPDs), can be directly generated using C++ [30], or Java [61] constructs.

6.3.2 Synchronization policy / Life-time analysis

To derive a synchronization policy used to exchange data between a directly connected producer/consumer set of processes, we need to perform a *lifetime analysis* for each of the communicated data tokens. That is, a consumer process has to be able to detect when the token stored at a certain memory location in a communication channel is consumed for the last time. After that the validity bit is set to zero ($vb = 0$), acknowledging the producer process that overwriting the considered memory location is safe. In case the communication is without multiplicity, the overwriting policy is trivial, i.e., after the consumption of the token the producer is free to overwrite. In case multiplicity is involved the consumer has to be able to determine on it self what is the last consumption (within a group of consumptions of the same token) of a produced token, such that only after the last consumption took place the validity bit is set on zero. As further shown in Section 6.3.3, in case of IOM+ communication, instead of looking at the last consumption of a token within a group of consecutive consumptions, it is useful to detect the first consumption.

For the P/C pair shown in Figure 6.5, each firing of the consumer process inside the domain $C' = \{(i,j) \in R^2 \mid j = 4, 1 \leq i \leq 4\}$ corresponds to an iteration point at which a token is consumed for the last time. Hence, the validity bit associated to that memory location should be set to zero. For the same P/C pair the token is consumed for the first time at iterations inside the domain $c' = \{(i,j) \in R^2 \mid i = j, 1 \leq i \leq 4\}$. As shown later we use this set when implementing the communication for IOM+ channels.

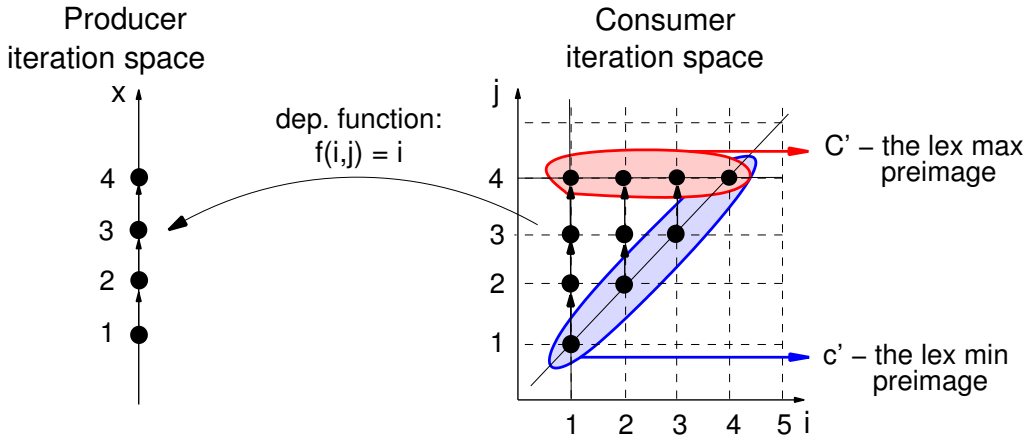


Figure 6.5: Lifetime analysis over a P/C pair with multiplicity

Problem Definition

In case of multiplicity, the synchronization policy will be implemented by deriving in a systematic way (at compile time) the lexicographic minimal and maximal domains. Those domains will be expressed (synthesized) in terms of the consumer iteration space iterators. Further on we introduce these two domains in a formal way.

Consider a P/C pair of a PN with a consumer input port domain represented by the parameterized polytope $\mathcal{C}(N) \in R^n$. According to equation 4.1, the producer domain is defined as the following linearly bounded lattice:

$$OPD = f(\mathcal{C}(N) \cap Z^n) = \{y \mid y = f(x) \ \wedge \ \exists\, x \in \mathcal{C}(N) \cap Z^n\}, \tag{6.1}$$

where f is an affine dependency function. For each $y \in OPD$ consider the set:

$$f^{-1}(y) = \{x \in \mathcal{C}(N) \cap Z^n \mid f(x) = y\}. \tag{6.2}$$

Let $f_M^{-1}(y)$ and $f_m^{-1}(y)$ denote the lexicographic maximal respectively minimal element in the set $f^{-1}(y)$:

$$f_M^{-1}(y) \quad = max_{lex}\{x \in \mathcal{C}(N) \cap Z^n \mid f(x) = y\}, \tag{6.3}$$
$$f_m^{-1}(y) \quad = min_{lex}\{x \in \mathcal{C}(N) \cap Z^n \mid f(x) = y\}. \tag{6.4}$$

$f_M^{-1}(y)$ will be referred to as the *lexicographic maximal preimage* of y and $f_m^{-1}(y)$ as the *lexicographic minimal preimage* of y.

Consider the sub-domains $C' \subseteq OPD$ and $c' \subseteq OPD$ which consist of all maximal preimages: $C' = \{f_M^{-1}(y) \mid y \in f(\mathcal{C}(N) \cap Z^n)\}$ respectively all minimal preimages $c' = \{f_m^{-1}(y) \mid y \in f(\mathcal{C}(N) \cap Z^n)\}$. We say that C' and c' are the *Lexicographically Maximal Preimage (LMP)* domain respectively *Lexicographically Minimal Preimage (LmP)* domain of polytope C through affine function f.

With respect to the life-time analysis, the set C' has the following meaning. Let t be a token produced by the producer IP $y \in OPD$ and stored at memory location $write(y)$. Then, C' contains

$f_M^{-1}(y)$, the lexicographically maximal consumer iteration point that consumes t. This means that $f_M^{-1}(y)$ is the last ip consuming t. Similarly $f_m^{-1}(y)$ is the first ip consuming t. *Therefore, a token t produced at IP y necessarily has to be alive in the memory between the firing of $f_m^{-1}(y)$ and $f_M^{-1}(y)$.*

According to the definition, C' contains all such last consuming IPs, and only them. Hence, if a consumer IP y belongs to C', then once y has fired, the memory location $read(f_M^{-1}(y))$ can be released and further rewritten. On the other hand, if a consumer ip y belongs to c', then once y has fired, the memory location $read(f_m^{-1}(y))$ is accessed for the first time.

In the coming two sections we present two procedures to derive the lexicographically maximal/minimal preimages and their domains of existence, followed by a number of practical examples.

Solution - ILP approach

This section presents how the Lexicographically Maximal/Minimal Preimage can be obtained using parametric integer linear programming. Relation (6.3) can be recast in the following parametric ILP problem (,where x and N are variables and y is parameter):

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{C}(N), \\ & y = f(x), \\ \textbf{objective:} \quad & x_m = max_{\textbf{lex}}\{x(y)\}, \end{aligned} \tag{6.5}$$

of which multi-stage solution represents $f_M^{-1}(y)$:

$$f_M^{-1}(y) = \begin{cases} T_1(y), \text{ if } y \in \mathcal{D}_1, \\ : \\ T_n(y), \text{ if } y \in \mathcal{D}_n. \end{cases} \tag{6.6}$$

Similarly, by setting the objective of the previous ILP program to do the lexicographic minimum we get $f_m^{-1}(y)$. By performing the integral image of each domain D_i through its corresponding function T_i, where $T_i \neq \perp$, we get as a result a number of linearly bounded latices:

$$C_i = T_i(\mathcal{D}_i \cap Z^m). \tag{6.7}$$

The union of those LBLs restricted to $\mathcal{C}(N)$ represents the lexicographically maximum preimage of $\mathcal{C}(N)$ through the function f:

$$C' = \bigcup_{i \in I} (C_i \cap \mathcal{C}(N)) \text{ , where } I = \{i \mid T_i \neq \perp\}. \tag{6.8}$$

However, C' is not expressed in terms of the consumer iterators, and therefore, can not be used to determine memory release moments. To express C' in terms of the consumer iterators, we can make use of the procedure used for filtering holes presented in Chapter 3. This means that *the possible holes that can be introduced when constructing the real image of C_i through a non-unimodular mapping T_i have to be filtered out* by placing adequate constraints (see Example 3 from Section 6.3.2). As a

general procedure for each of the LBLs we formulate an ILP problem similar to the one used to derive an OPD in Section 4.2:

$$\begin{aligned} \textbf{subject to:} \quad & t \in \mathcal{D}_i, \\ & v \in T_i(D_i), \\ & v = T_i(t), \\ \textbf{objective:} \quad & t_M = max_{\textbf{lex}}\{t(v)\}. \end{aligned} \tag{6.9}$$

The solution of this problem is a multistage conditional expression from which only the non-empty branches correspond to lexicographically maximal preimage points:

$$\begin{aligned} \text{if}(v \in \ \mathcal{S}_{i,1}), & \\ \text{then} \quad & t_M = L_{i,1}(v), \\ \text{else if}(v \in \ \mathcal{S}_{i,2}), & \\ \text{then} \quad & t_M = L_{i,2}(v), \\ & \vdots \\ \text{else if}(v \in \ \mathcal{S}_{i,n}), & \\ \text{then} \quad & t_M = L_{i,n}(v). \end{aligned}$$

Therefore,

$$C_i \cap \mathcal{C}(N) = \bigcup_{j \in J_i} \mathcal{S}_{i,j}, \text{ where } J_i = \{j \mid L_{i,j} \neq \perp\}, \tag{6.10}$$

such that by combining equation 6.10 into equation 6.8 we finally get the lexicographical maximal domain in terms of the consumer iterators:

$$C' = \bigcup_{i \in I} \bigcup_{j \in J_i} \mathcal{S}_{i,j}. \tag{6.11}$$

If for a certain i the above solution does not contain any empty branch i.e. $\forall\ j,\ L_{i,j} \neq \perp$, then the corresponding preimage C_i can be obtained as a real image by replacing equation 6.7 with:

$$C_i = T_i(\mathcal{D}_i). \tag{6.12}$$

Examples

We illustrate the presented technique as it applies to three P/C pairs. Because the first part of the life time analysis procedure coincides with the procedure used to construct the producer domain, the presented examples can be considered as examples for the Production Restructuring step as well.

Example 1: Consider the following P/C program in which the producer communicates data to the consumer through an unidimensional array `a[]`:

```
for (int x=1; x<=4; x++){
    a[x] = Fp();          // Producer
}
for (int i=1; i<=4; i++){
  for (int j = 1; j<=i, j++){
    Fc( a[i] );          // Consumer
  }
}
```

The iteration spaces of the producer and consumer are graphically depicted in Figure 6.6. The Consumer domain is represented by the integral points inside the polytope:

$$\mathcal{C} = \{(i,j) \in R^2 \mid 1 \leq i \leq 4 \wedge i \leq j \leq 4\}. \tag{6.13}$$

The affine transformation $f(i,j) = i$ generates the following producer domain (in this case the real image does not introduce holes):

$$\mathcal{P} = \{x \in R \mid 1 \leq x \leq 4\}. \tag{6.14}$$

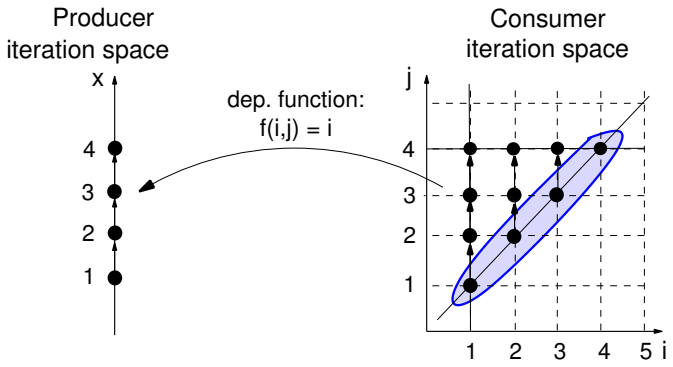


Figure 6.6: The Lexicographically Minimal Preimage for a IOM+ P/C pair

To extract the *Lexicographically Minimal Preimage* (LmP), we follow the presented approach but change the objective function of problem 6.5 into $x_m = min_{\mathbf{lex}}\{x(y)\}$. For the considered P/C pair, computing $LmP(x)$ is equivalent to solving the following PIP problem with parameters (i,j):

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq \mathrm{i} \leq 4, \\ & i \leq \mathrm{j} \leq 4, \\ & x = \mathrm{i}, \\ \textbf{objective:} \quad & min_{\mathbf{lex}}\{(i,j)\}. \end{aligned} \tag{6.15}$$

This PIP problem has the following solution:

```
1.  if(0 <= -x+4),
2.     if( 0 <= x-1),
3.          (i,j)=(x,x);
4.     end
5.  end
```

This solution tree gives an affine function $T(x) = (x,x)$ (see line 3 in the previous pseudo-code) that is defined on the domain: $\mathcal{P}_1 = \{x \in Z \mid 1 \leq x \leq 4\}$. Because no empty branch in the second PIP problem can be found, we conclude that $T(x)$ does not introduce holes at the consumer side. Thus, the real image of $\mathcal{P}_1$ through $T(x)$ represents the LmP set:

$$c' = \{(i,j) \in R^2 | 1 \leq i \leq 4 \wedge i = j\}. \tag{6.16}$$

Example 2: Consider the following P/C pair:

```
for (int j=0; j<=3*N+3; j++){
    a[j] = Fp();          // Producer
}
for (int k1=0; k1<=N; k1++){
  for (int k2 = k1; k2<=N, k2++){
    Fc( a[2*k1+k2+3] );  // Consumer
  }
}
```

Choosing the program parameter $N = 5$, the iteration spaces of producer and consumer are as depicted in Figure 6.7. The Consumer domain (corresponding to function Fc) is described by the integer points

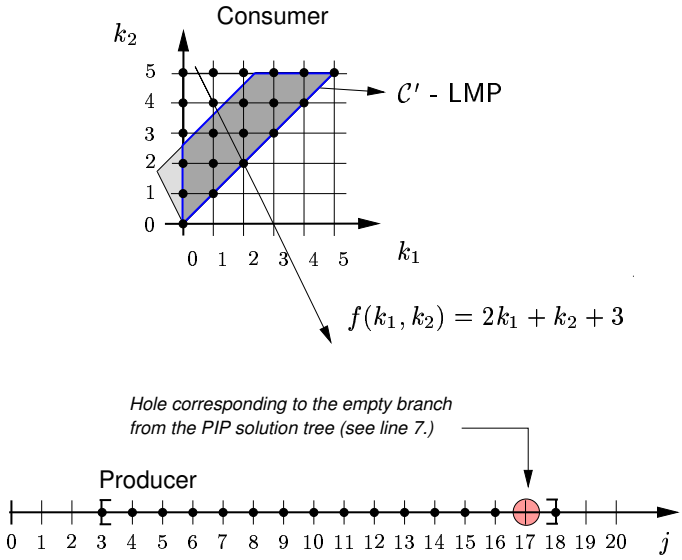


Figure 6.7: P/C iteration spaces

inside the following parameterized polytope:

$$\mathcal{C}(N) = \{(k_1, k_2) \in R^2 \mid 0 \leq k_1 \leq N \wedge k_1 \leq k_2 \leq N\}. \tag{6.17}$$

Performing the real image of this domain through the affine transformation $f(k_1, k_2) = 2k_1 + k_2 + 3$, we get the following polyhedral domain:

$$\mathcal{P} = \{x \in R \mid 3 \leq x \leq 3N + 3\}. \tag{6.18}$$

Although the real image is able to filter some of the producer points for which corresponding consumption does not exist, the resulted domain, i.e., $\mathcal{P}$ is different from the producer OPD as it contains an extra iteration point $j = 17$ (see Figure 6.7). Thus, to obtain the proper OPD, the following ILP (instance of 6.5) has to be solved:

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq k_1 \leq N, \\ & k_1 \leq k_2 \leq N, \\ & \mathrm{j} = 2k_1 + k_2 + 3, \\ \textbf{objective:} \quad & max_{\textbf{lex}}\{(k_1, k_2)\}. \end{aligned} \tag{6.19}$$

The solution to this problem is represented by the following multi-cast pseudo code with only one non-empty branch:

```
1. if (0 <= j-3),
2.     if(0 <= -j+3*N+3),
3.       _d18 = div(2*j,6),
4.           if(0 <= -j+N+2*_d18+1),
5.               (k1,k2)=(_d18-1,j-2*_d18-1);
6.           else
7.               Nil;
8.           end
9.     else
10.       Nil;
11.    end
12. else
13.     Nil
14. end.
```

This solution tree is a LBL representing the Producer OPD. This LBL represents the integer points embedded in the following two dimensional polytope with coordinates j and d_{18}:

$$\mathcal{P}(j, d_{18}) = \{(j, d_{18}) \in R^2 \mid 3 \leq j \leq 3N + 3 \ \wedge \ 0 \leq 2j - 6d_{18} \leq 5 \ \wedge \ 0 \leq -j + 2d_{18} + N + 1\}.$$

As shown in Figure 6.7, for $N = 5$ the Producer iteration $j = 17$ is filtered out such that data is not sent to the Consumer. Line 5 in the pseudo-code, indicates the mapping function $T(j, d_{18}) = (d_{18} - 1, j - 2d_{18} - 1)$. By performing the real image of $\mathcal{P}$ through T we get the following polyhedral domain:

$$T(\mathcal{P}(N)) = \{(k_1, k_2) \in R^2 \ \mid \ 0 \leq -k_1 + k_2 \ \wedge \ k_2 \leq N \ \wedge \ 0 \leq 2k_1 - 2k_2 + 5 \ \wedge \ 0 \leq 2k_1 + k_2\}.$$

Next, we get the lexicographically maximal preimage domain by solving (corresponding to problem 6.9) the following PIP problem:

$$\begin{aligned} \textbf{subject to:} \quad & 3 \leq j \leq 3N + 3 \\ & 0 \leq 2j - 6d_{18} \leq 5 \\ & 0 \leq -j + 2d_{18} + N + 1 \\ & 0 \leq -k_1 + k_2 \\ & k_2 \leq N \\ & 0 \leq 2k_1 - 2k_2 + 5 \\ & 0 \leq 2k_1 + k_2 \\ & k_1 = d_{18} - 1, \\ & k_2 = j - 2d_{18} - 1 \\ \textbf{objective:} \quad & max_{\textbf{lex}}\{(k_1, k_2)\}. \end{aligned} \tag{6.20}$$

The solution to this problem has no empty branch. Similarly to the previous example, based on equation 6.12 we conclude that the lexicographically maximum preimage domain is given by the real image of $\mathcal{P}(N)$ through T: $C' = T(\mathcal{P}(N))$. Although C' is defined by four inequalities, three of them are redundant with the boundaries of the consumer for-loops. Only one inequality will have to be added to the consumer specification to determine the lexicographically maximal preimage domain

(i.e., $0 \leq 2k_1 - 2k_2 + 5$).

Example 3: Consider the following P/C program in which the producer communicates data to the consumer through an unidimensional array `a[]`:

Producer:

```
for (i = 0; i<=10;i++){
    a[i] =Fp(...);
}
```

Consumer:

```
for (x =0; x<=11; x++){
 for (y =0; y<=4; y++){
  for (z = 0; z<=10; z++){
   if (x-4*y+3>=0){
    if (-x+z+1>=0){
     if (2*x+4*y-3*z-3>=0){
      ...=Fc(a[z]);
     }
    }
   }
  }
 }
}
```

Our Compaan compiler classifies this example as an IOM+ P/C pair. Due do the multiplicity, we have to determine the lifetime of the data communicated over the array `a[]`. The iteration spaces of the producer and consumer are graphically depicted in Figure 6.8. For another 3D sketch of the P/C iteration spaces and the dependency relations, we refer to Figure 6.9.

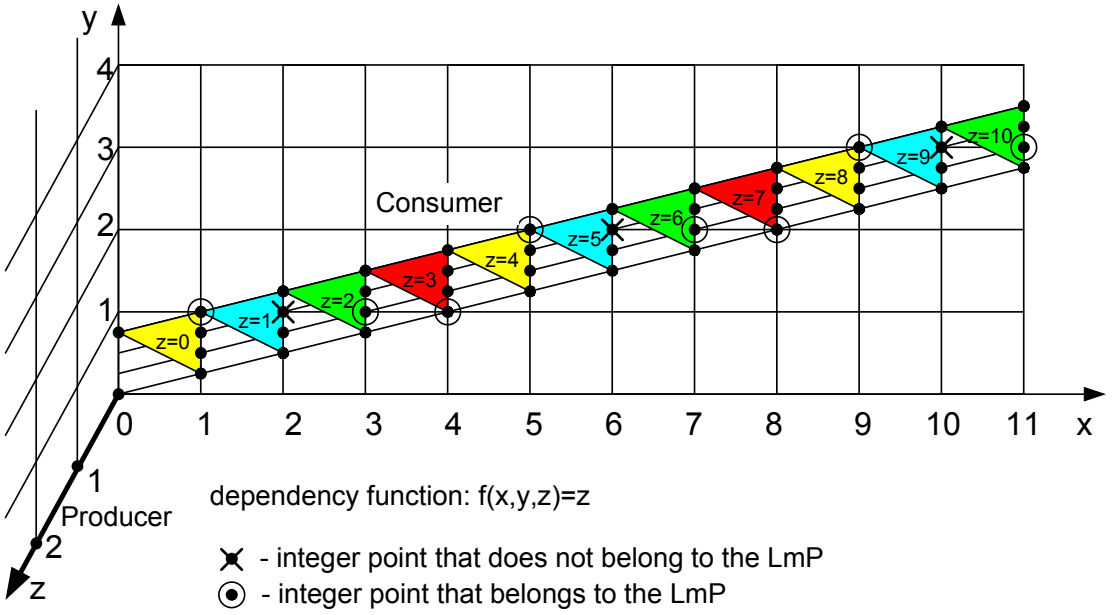


Figure 6.8: Consumer and Producer domains for P/C pair of Example 3. ILP is needed for LmP calculation.

The Consumer domain is described by the integral points inside the next non-parametric polytope:

$$\mathcal{C} = \{(x,y,z) \in R^3 \mid \quad 0 \leq x \leq 11 \wedge 0 \leq y \leq 4 \wedge 0 \leq z \leq 10 \wedge x - 4*y + 3 >= 0 \wedge \\ -x + z + 1 >= 0 \wedge 2*x + 4*y - 3*z - 3 >= 0\}.$$

To get the producer OPD the following ILP (instance of 6.5) has to be solved:

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq x \leq 11, \\ & 0 \leq y \leq 4, \\ & 0 \leq z \leq 10, \\ & x - 4*y + 3 \geq 0, \\ & -x + z + 1 \geq 0, \\ & 2*x + 4*y - 3*z - 3 \geq 0, \\ & i = z, \\ \textbf{objective:} \quad & min_{\mathbf{lex}}\{(x,y,z)\}. \end{aligned} \tag{6.21}$$

The solution is represented by a multi-cast expression with one non-empty branch:

```
1.  if(0<=z<=10){
2.       d1 = div(9*z+3,12);
3.       d2 = div(7*z+4*d1+9,10);
4.       d3 = div(z+2*d2+3,4);
5.       (x,y,z)=(2*z-d2+1,d2-d3+1,z);
6.  } else if(z>=11){
7.      Nil;
8.  }
```

This solution tree is a polytope that represents the Producer domain: $\mathcal{P} = \{j \in R^2 \mid 0 \leq j \leq 10\}$. In this case, by performing the real image of $\mathcal{P}$ through T, we get the following polyhedral domain:

$$T(\mathcal{P}) = \{(x,y,z) \in R^3 \mid \quad 0 \leq x \leq 11 \wedge 0 \leq y \leq 4 \wedge 0 \leq z \leq 10 \wedge x - 4*y + 3 >= 0 \wedge \\ -x + z + 1 >= 0 \wedge 2*x + 4*y - 3*z - 3 >= 0\}.$$

Observe that $T(\mathcal{P}) = \mathcal{C}$. Because we have an IOM+ communication type, we cannot use $T(\mathcal{P})$ as the LmP domain. This phenomena is different from the previous two examples where the LmP/LMP domain was identical to the real-image of the OPD through the LmP/LMP function. We get the lexicographically maximal preimage domain by solving (corresponding to problem 6.9) the following PIP problem:

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq z \leq 10, \\ & 0 \leq 9*z + 3 - 12*d_1 \leq 11, \\ & 0 \leq 7*z + 4*d_1 + 9 - 10*d_2 \leq 9, \\ & 0 \leq z + 2*d_2 + 3 - 4*d_3 \leq 4, \\ & (x,y,z) = (2*z - d_2 + 1, d_2 - d_3 + 1, z); \\ \textbf{objective:} \quad & min_{\mathbf{lex}}\{(x,y,z)\}. \end{aligned} \tag{6.22}$$

The solution to this problem looks quite complex and contains two non-empty branches:

```
if (-2*x-4*y+3*z+6)
     if (-z+10)
        if (x-z)
           if (2*x+4*y-3*z-3)
              if (-15*x+15*z+19)
                 if (-10*x+10*z+9)
                     d7 = div(9*z+3,12)
                       if (10*x+4*d7-1)
                          if (4*x-52*y+12*d7+75)
                             if (10*x-13*z+4*d7-1)
                                  d8 = div(10*x+4*d7+12,13)
                                  d9 = div(4*d7+7*d8+2,10)
                                   Sol: (d8-1,d7,d9,-y+d9+1)
                             else (-10*x+13*z-4*d7)
                                Nil
                          else (-4*x+52*y-12*d7-76)
                             Nil
                       else (-10*x-4*d7)
                          Nil
                 else (10*x-10*z-10)
                     d10 = div(2*x+z+2,4)
                       if (-36*x+27*z+12*d10+32)
                           Sol: (z,-3*x+3*z+d10+2,-x+2*z+1,-x-y+2*z+2)
                       else (36*x-27*z-12*d10-33)
                          Nil
              else (15*x-15*z-20)
                 Nil
           else (-2*x-4*y+3*z+2)
              Nil
        else (-x+z-1)
           Nil
     else (z-11)
        Nil
  else (2*x+4*y-3*z-7)
     Nil
```

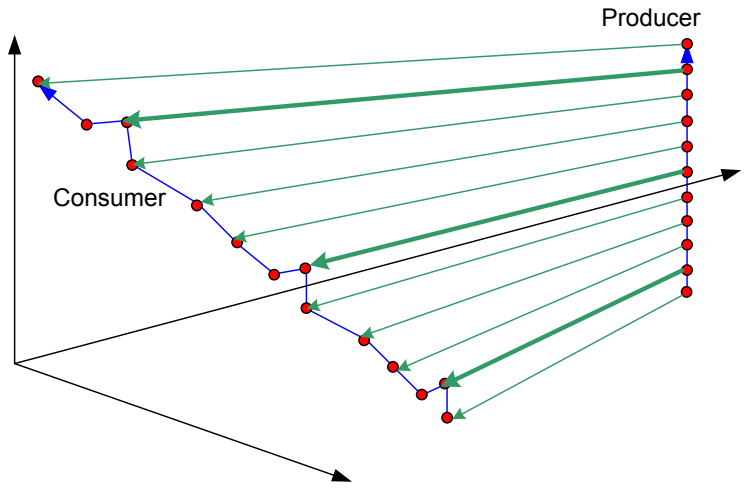


Figure 6.9: 3-D visualization of the P/C pair of Example 3; the producer domain is exposed in the back part of the picture, while the consumer is in the front. The thick horizontal lines show the multiplicity structure.

The tree exposes two solutions (starting with **Sol**) which together represent the lexicographically minimal preimage of the P/C pair. Each time when a Consumer iteration validates one of the solutions of this problem then data has to be read from the input FIFO buffer (for more information regarding the synthesis of the communication channel we refer to Section 6.3.3). Based on the LmP definition, we obtain the following PN representation where the producer process is:

Producer:

```
for ( int z = 0 ; z <=10; z += 1 ) {
  out=Fp(...);
  FIFO.put(out);
}
```

and the consumer process is:

Consumer:

```
for (int x = ceil(0) ; x <= (int) floor(11 ); x +=1) {
 for (int y = ceil(Math.max(x/4,(-2*x+3)/4);y<=floor((1*x+3)/4);y +=1) {
  for (int z = ceil(Math.max(0,1*x+-1));z<=floor(Math.min((2*x+4*y+-3)/3,10); z+=1){
          int div7 =(int)div(9*z+3,12);
          int div8 =(int)div(10*x+4*div7+12,13);
          int div9 =(int)div(4*div7+7*div8+2,10);
          int div10 =(int)div(2*x+1*z+2,4);
          if ( -2*x+-4*y+3*z+6 >= 0) {
            if ( -1*z+10 >= 0) {
              if ( 1*x+-1*z >= 0) {
                if ( 2*x+4*y+-3*z+-3 >= 0) {
                  if ( -15*x+15*z+19 >= 0) {
                    if ( -10*x+10*z+9 >= 0) {
                      if ( 10*x+4*div7+-1 >= 0) {
                        if ( 4*x+-52*y+12*div7+75 >= 0) {
                          if ( 10*x+-13*z+4*div7+-1 >= 0) {
                            in = FIFO.get();
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
          if ( -2*x+-4*y+3*z+6 >= 0) {
            if ( -1*z+10 >= 0) {
              if ( 1*x+-1*z >= 0) {
                if ( 2*x+4*y+-3*z+-3 >= 0) {
                  if ( -15*x+15*z+19 >= 0) {
                    if ( 10*x+-10*z+-10 >= 0) {
                      if ( -36*x+27*z+12*div10+32 >= 0) {
                        in = FIFO.get();
                      }
                    }
                  }
                }
              }
            }
          }
          ...= Fc(in) ;
        } // for z
      } // for y
    } // for x
  }
```

This PN representation seams to be control expensive. If you look in Figure 6.8, you can see that actually the LmP domain consists of the integral points along 3 parallel lines. Thus, we may ask how can we formally obtain these three lines representing the LmP. In this context, the next section presents an alternative analytical procedure that allows to obtain the preimage domain (LmP or LMP) as a union of polytopes. In case this union is composed of a small number of polytopes, then the LMP/LmP formulation becomes less expensive than the current one.

Another solution for constructing LMP/LmP

The method used in Section 6.3.2 for deriving the LMP and LmP domains makes excessive use of linear programming formulations. This may lead to a quite complex representation of the LMP domain (see Example 3 from the previous section). In this section, we present an alternative technique for deriving the LMP/LmP domains, that avoids solving the ILP problems given by 6.9. Consider again the ILP problem used to derive the LMP domain:

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{C}(N), \\ & y = f(x), \\ \textbf{objective:} \quad & x_m = max_{\textbf{lex}}\{x(y)\}, \end{aligned} \tag{6.23}$$

with it is multi-cast solution:

$$f_M^{-1}(y) = \begin{cases} T_1(y), \text{ if } y \in \mathcal{D}_1 \\ : \\ T_n(y), \text{ if } y \in \mathcal{D}_n. \end{cases} \tag{6.24}$$

Compared to the real image $f(\mathcal{C}(N))$, the domains $\mathcal{D}_i$ include extra dimensions represented by div statements. In the solution of problem 6.19, an extra dimension $d_{18} = div(2j, 6)$ is introduced (see line 3 of the solution tree). Based on this observation two definitions are further introduced:

Expansion functions

Definition 6 *Corresponding to an arbitrary domain $\mathcal{D}_i$ obtained by solving an LMP/LmP problem (as given by 6.23), the* producer expansion function $PEF_i : \mathcal{D}_i \cap Z \to Z$ *is defined as follows:*

$$PEF_i(y) = (y, d_1 = div_1(f_1(y), t_1), d_2 = div_2(f_2(y, d_1), t_2), ..., d_n = div_n(f_n(y, d_1, ..., d_{n-1}), t_n)),$$

where div_i represents the "div" operator introduced by the ILP solution characterized by a linear function f_i, and a constant t_i.

Example: Here we present the PEF functions for the three examples given in the previous section:

- for Example 1, there is a unique PEF is $PEF(j) = (j, d_1 = div(2j, 6))$.
- for Example 2, the PEF is the identity function: $PEF(x) = x$.
- for Example 3, there is again only one expansion function $PEF(z) : [0, 10] \to Z^4$:

  $PEF(z) = (z, d_1 = div(9z + 3, 12), d_2 = div(7z + 4d_1 + 9, 10), d_3 = div(z + 2d_2 + 3, 4)).\square$

We use the PEF function, to define the expansion of a producer domain:

Definition 7 *Corresponding to an arbitrary domain $\mathcal{D}_i$ obtained by solving an LMP/LmP problem (as given by 6.23), the* producer expanded domain *PED represents the following set:*

$$\mathcal{P}_i^E = \{(y, d_1, ..., d_i, ...) | y \in f(\mathcal{C}), 0 \leq f_1(y) - d_1 t_1 \leq t_1 - 1, 0 \leq f_2(y, d_1) - d_2 t_2 \leq t_2 - 1, \\ ..., 0 \leq f_i(y, d_1, ..., d_{i-1}) - d_i t_i \leq t_i - 1, ...\}.$$

Observe that, due to the added Gomory cuts, $\mathcal{D}_i \subset \mathcal{P}_i^E$ (see line 4. in the solution to the problem given in 6.19, and visualized in the left part of Figure 6.10).

Example: For the problem given in 6.19, where the consumer is the triangle $\mathcal{C}(N) = \{(k_1, k_2) \in R^2 \mid 0 \leq k_1 \wedge k_1 \leq k_2 \leq N\}$ and the dependency function is the projection $f(k_1, k_2) = 2k_1 + k_2 + 3$, the PED is: $\mathcal{P}_1^E(N) = \{(\{(j, d_{18}) \in R^2 \mid 3 \leq j \leq 3N + 3 \ \wedge \ 0 \leq 2j - 6d_{18} \leq 5\}$. □

Here we introduce an alternative representation of the producer expansion function. This representation is based on the observation that the "div" operator can be expressed as: $div(a, b) = \frac{a - a\%b}{b}$, where the modulo operator, i.e., $a\%b$ is the reminder of the integer division $\frac{a}{b}$. Hence,

$$d_1 = \begin{cases} \frac{f_1(y)}{t_1} & \text{if } f_1(y)\%t_1 = 0, \\ \frac{f_1(y)-1}{t_1} & \text{if } f_1(y)\%t_1 = 1, \\ : & \\ \frac{f_1(y)-t_1+1}{t_1} & \text{if } f_1(y)\%t_1 = t_1 - 1, \end{cases} \qquad d_2 = \begin{cases} \frac{f_2(y,d_1)}{t_2} & \text{if } f_2(y, d_1)\%t_2 = 0, \\ \frac{f_2(y,d_1)-1}{t_2} & \text{if } f_2(y, d_1)\%t_2 = 1, \\ : & \\ \frac{f_2(y,d_1)-t_2+1}{t_2} & \text{if } f_2(y, d_1)\%t_2 = t_2 - 1, \end{cases}$$

$$\ldots \qquad d_n = \begin{cases} \frac{f_n(y,d_1,...,d_{n-1})}{t_n} & \text{if } f_n(y, d_1, ..., d_{n-1})\%t_n = 0, \\ \frac{f_n(y,d_1,...,d_{n-1})-1}{t_n} & \text{if } f_n(y, d_1, ..., d_{n-1})\%t_n = 1, \\ : & \\ \frac{f_n(y,d_1,...,d_{n-1})-t_n+1}{t_n} & \text{if } f_n(y, d_1, ..., d_{n-1})\%t_n = t_n - 1. \end{cases}$$

Each possible combination $(k_1 = f_1(y)\%t_1, k_2 = f_2(y, d_1)\%t_2, ..., k_m = f_n(y, d_1, ..., d_{n-1})\%t_n)$ of values of the modulo functions defines the following lattice:

$$L_k^P = \{y \in Z \mid f_1(y)\%t_1 = k_1, f_2(y, d_1)\%t_2 = k_2, ..., f_n(y, d_1, ..., d_{n-1})\%t_n = k_m\}. \tag{6.25}$$

Restricted to a particular lattice L_k^P, the producer expansion function PEF becomes a linear function. For example, consider the lattice L_0^P containing the integral points y with the property that, $f_1(y)\%t_1 = 0 \wedge f_1(y, d_1)\%t_1 = 0 \wedge ...f_n(y, d_1, ..., d_{n-1})\%t_n = t_n - 1$. Thus, if $y \in L_0^P$ then:

$$PEF_0(y) = (y, d_1 = \frac{f_1(y)}{t_1}, d_2 = \frac{f_2(y, d_1)}{t_2}, ..., d_n = \frac{f_n(y, d_1, ..., d_{n-1})}{t_n}).$$

There are $m = t_1 t_2 ... t_n$ possible combinations of the modulo expressions. This implies that the producer space (restricted to integer points) is covered by m (spanning) lattices and to each lattice corresponds a constant rational vector $V^k = (v_1^k, v_2^k, ..., v_n^k)$. Based on the spanning lattices we can rewrite the PEF function as follows:

$$PEF_i(y) = (y, d_1 = \frac{f_1(y)}{t_1}, d_2 = \frac{f_2(y, d_1)}{t_2}, ..., d_n = \frac{f_n(y, d_1, ..., d_{n-1})}{t_n}) + \begin{cases} V^0, \text{if } y \in L_0^P, \\ V^1, \text{if } y \in L_1^P, \\ : \\ V^m, \text{if } y \in L_m^P. \end{cases} \tag{6.26}$$

We call this form of expressing the PEF_i function, the lattice representation of PEF_i.

Example: Here we present the lattice representation of the PEF functions for the three examples given in the previous paragraph:

- for Example 1, there are three lattices so the PEF can be expressed as follows:

$$PEF(j) = (j, \frac{j}{3}) + \begin{cases} (0,0) & \text{if } j\%3 = 0, \\ (0, \frac{-1}{3}) & \text{if } j\%3 = 1, \\ (0, \frac{-2}{3}) & \text{if } j\%3 = 2. \end{cases}$$

- for Example 2, there are no div operators involved in the ILP solution, thus, no lattices. Hence, the PEF function is: $PEF(x) = x, \forall x$.
- for Example 3, where the initial representation of the PEF functions is:

  $PEF(z) = (z, d_1 = div(9*z+3, 12), d_2 = div(7*z+4*d_1+9, 10), d_3 = div(z+2*d_2+3, 4))$.

  we get the following lattice representation of the PEF:

$$PEF(z) = (z, \frac{9*z+3}{12}, \frac{7*z+4*\frac{9*z+3}{12}+9}{10}, \frac{z+2*\frac{7*z+4*\frac{9*z+3}{12}+9}{10}+3}{4}) +$$

$$+ \begin{cases} \begin{cases} (0,0,0,0) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 0 \wedge (z+2*\frac{7*z+4*\frac{9*z+3}{12}+9}{10}+3)\%4 = 0, \\ (0,0,0,\frac{-1}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 0 \wedge (z+2*\frac{7*z+4*\frac{9*z+3}{12}+9}{10}+3)\%4 = 1, \\ (0,0,0,\frac{-2}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 0 \wedge (z+2*\frac{7*z+4*\frac{9*z+3}{12}+9}{10}+3)\%4 = 2, \\ (0,0,0,\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 0 \wedge (z+2*\frac{7*z+4*\frac{9*z+3}{12}+9}{10}+3)\%4 = 3, \end{cases} \\ + \begin{cases} (0,0,\frac{-1}{10},0) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 1 \wedge (z+2*\frac{7*z+4*\frac{9*z+2}{12}+9}{10}+3)\%4 = 0, \\ (0,0,\frac{-1}{10},\frac{-1}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 1 \wedge (z+2*\frac{7*z+4*\frac{9*z+2}{12}+9}{10}+3)\%4 = 1, \\ (0,0,\frac{-1}{10},\frac{-2}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 1 \wedge (z+2*\frac{7*z+4*\frac{9*z+2}{12}+9}{10}+3)\%4 = 2, \\ (0,0,\frac{-1}{10},\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 0 \wedge (7*z+4*\frac{9*z+3}{12}+9)\%10 = 1 \wedge (z+2*\frac{7*z+4*\frac{9*z+2}{12}+9}{10}+3)\%4 = 3, \end{cases} \\ \vdots \\ \begin{cases} (0,\frac{-2}{3},\frac{-9}{10},\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 2 \wedge (7*z+4*\frac{9*z+1}{12}+9)\%10 = 9 \wedge (z+2*\frac{7*z+4*\frac{9*z+1}{12}+9}{10}+3)\%4 = 0, \\ (0,\frac{-2}{3},\frac{-9}{10},\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 2 \wedge (7*z+4*\frac{9*z+1}{12}+9)\%10 = 9 \wedge (z+2*\frac{7*z+4*\frac{9*z+1}{12}+9}{10}+3)\%4 = 1, \\ (0,\frac{-2}{3},\frac{-9}{10},\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 2 \wedge (7*z+4*\frac{9*z+1}{12}+9)\%10 = 9 \wedge (z+2*\frac{7*z+4*\frac{9*z+1}{12}+9}{10}+3)\%4 = 2, \\ (0,\frac{-2}{3},\frac{-9}{10},\frac{-3}{4}) \text{ if } (9*z+3)\%3 = 2 \wedge (7*z+4*\frac{9*z+1}{12}+9)\%10 = 9 \wedge (z+2*\frac{7*z+4*\frac{9*z+1}{12}+9}{10}+3)\%4 = 3. \end{cases} \end{cases}$$

□

Because $y = f(x)$ where f is the dependency function, by replacing y in 6.26, it turns out that to each producer lattice L_k^P it corresponds a consumer lattice L_k^C.

$$L_k^C = \{x \in Z \mid f_1(f(x))\%t_1 = k_1, f_2(f(x), d_1)\%t_2 = k_2, ..., f_n(f(x), d_1, ..., d_{n-1})\%t_n = k_m\}.$$

According to this definition and in the context of our discussion, we obtain the following proposition:

Proposition 8 *If a and b are two consumer integer points such that $f(a) = f(b)$ then they both belong to the same consumer lattice L_k^C.*

Based on the notion of PEF the following function can be established.

Definition 8 *The* explicit dependency function *is defined as follows:* $f^E : \mathcal{C} \to \mathcal{P}_i^E$*:*

$$f^E(x) = \begin{cases} f_0^E(x) & \text{if } x \in L_0^{\mathcal{C}}, \\ f_1^E(x) & \text{if } x \in L_1^{\mathcal{C}}, \\ : & \\ f_m^E(x) & \text{if } x \in L_m^{\mathcal{C}}, \end{cases}$$

where for an arbitrary k*,*

$$f_k^E(x) = (f(x), d_1 = \frac{f_1(f(x))}{t_1}, d_2 = \frac{f_2(f(x), d_1)}{t_2}, ..., d_n = \frac{f_n(f(x), d_1, ..., d_{n-1})}{t_n}) + V_k. \quad (6.27)$$

Example: In this example we derive the explicit dependency function for Example 1 (the other two

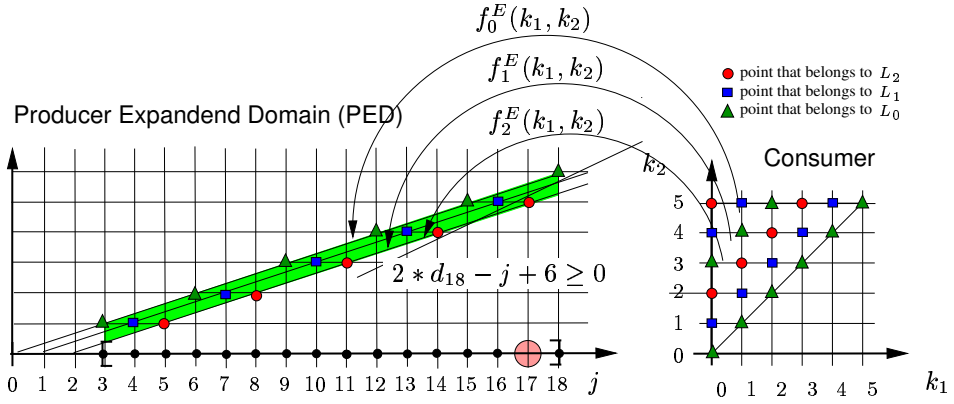


Figure 6.10: Making the dependency function explicit for each consumer lattice

examples are quite similar). In this case, there are three lattices $L_0^{\mathcal{C}} = \{(k_1, k_2) \in Z^2 \mid (2{*}k_1{+}k_2)\%3 = 0\}$, $L_1^{\mathcal{C}} = \{(k_1, k_2) \in Z^2 \mid (2 * k_1 + k_2)\%3 = 1\}$, and $L_2^{\mathcal{C}} = \{(k_1, k_2) \in Z^2 \mid (2 * k_1 + k_2)\%3 = 2\}$. The explicit dependency function is:

$$f^E(k_1, k_2) = \begin{cases} (2k_1 + k_2 + 3, \frac{2k_1+k_2+3}{3}) + (0, 0) & \text{if } (k_1, k_2) \in L_0^{\mathcal{C}}, \\ (2k_1 + k_2 + 3, \frac{2k_1+k_2+3}{3}) + (0, \frac{-1}{3}) & \text{if } (k_1, k_2) \in L_1^{\mathcal{C}}, \\ (2k_1 + k_2 + 3, \frac{2k_1+k_2+3}{3}) + (0, \frac{-2}{3}) & \text{if } (k_1, k_2) \in L_2^{\mathcal{C}}. \end{cases} \quad \square$$

To simplify our presentation, we restrict the original producer domain to an arbitrary domain $\mathcal{D} = \mathcal{D}_i$ on which a function $T = T_i$ is defined (see (6.24)). Consider the consumer subset $\{x \in \mathcal{C} \mid f(x) \in \mathcal{D}_i\}$ that is mapped by f into the domain $\mathcal{D}$; we call this set as $\mathcal{C}$. This leads to the following ILP problem:

$$\begin{aligned} \textbf{subject to:} \quad & x \in \mathcal{C}, \\ & y = f(x), \\ \textbf{objective:} \quad & x_m = max_{\textbf{lex}}\{x(y)\}. \end{aligned} \quad (6.28)$$

The solution function of this problem is now defined on a single domain:

$$f_M^{-1}(y) = T(y), \text{ if } y \in \mathcal{D}. \tag{6.29}$$

Next we analyze the properties of the function $G(x) = T \circ f^E(x)$. According to Definition 8, we analyze one of the component functions: $G_k(x) = T \circ f_k^E(x)$. We say that G_k is a *lattice maximal function*. According to (6.28) and (6.29) it turns out that:

Proposition 9 $x \in L_k^{\mathcal{C}} \Rightarrow G_k(x) = max_{\mathbf{lex}}\{x \in \mathcal{C} \mid f(x) = f(G_k(x))\}$.

Proposition 10 *Consider that $G_k(x) = T \circ f_k^E(x)$ is written in matrix form as $G_k(x) = Ax + B_k$, where A is a $(t \times t)$ square integral matrix and B_k is a $(t \times 1)$ rational offset. Then, $A^2 = A$ and $AB_k = 0_t$, where 0_t is a zero column matrix.*

Observe that while A is common to all the G_i functions, B_k is specific to G_k.

Proof: Consider that $x \in L_k^{\mathcal{C}}$. According to Proposition 9 we have out that $G_k(G_k(x)) = G_k(x)$. Hence, $A(Ax + B_k) + B_k = Ax + B_k \Rightarrow (A^2 - A)x + AB_k = 0_t$. But the elements (i.e., integer points) of the lattice L_k cover the whole space (according to Section 5.6, we assume a fully dimensional consumer domain), and therefore, $A^2 = A$ which further implies that $AB_k = 0_t$.

Theorem 6 *Let Π_k be the hyperplane defined by the equation: $G_k(y) = y$. Then,*

- *a. $\forall x \in L_k^{\mathcal{C}} \cap \mathcal{C}$ then $G_k(x) \in \Pi_k$, and*
- *b. $\forall y \in \Pi_k \cap \mathcal{C} \cap Z$, $\exists x \in L_k^{\mathcal{C}} \cap \mathcal{C}$ such that $y = G_k(x)$.*

Proof: a. If $x \in L_k^{\mathcal{C}} \cap \mathcal{C}$, then according to Proposition 10 it turns out that $G_k(G_k(x)) = G_k(x)$. Hence, $G_k(x) \in \Pi_k$.

b. Assume that $\exists y \in \Pi_k \cap \mathcal{C} \cap Z$ for which $\nexists x \in L_k \cap \mathcal{C}$ such that $y = G_k(x)$. This implies that y belongs to a lattice $L_p^{\mathcal{C}}$ which is different from $L_k^{\mathcal{C}}$ (otherwise suppose $y \in L_k \cap \mathcal{C}$ then $\exists x = y$ such that $y = G_k(x)$). Let G_p correspond to the lattice $L_p^{\mathcal{C}}$. Then, $f(y) = f(G_p(y))$, and $y \prec G_p(y)$ (y strictly precedes $G_p(y)$ because $L_k \neq L_p \Rightarrow B_k \neq B_p \Rightarrow G_k(y) \neq G_p(y) \Rightarrow y \neq G_p(y)$. Hence, $f(y) = f(Ay + B_p)$ (1) and $y \prec Ay + B_p$ (2).

On the other hand $y \in \Pi_k \Rightarrow Ay = y - B_k$ (3). (3) in (1) implies $f(y) = f(y + B_p - B_k)$, which further implies $\forall y, f(y) = f(y + B_p - B_k)$ (4).

(3) in (2) implies $y \prec y + B_p - B_k$ which further implies $0 \prec B_p - B_k$ (5).

Moreover, $y \in L_p \Rightarrow Ay + B_p \in Z$ and $y \in \Pi_k \Rightarrow Ay = y - By$ imply $y - B_k + B_p \in Z$. Hence, because $y \in Z$ it turns out that $B_p - B_k \in Z$ (6).

Altogether (4),(5) and (6) imply that $\forall y \in \Pi_k \cap \mathcal{C} \cap Z$, $\exists d \in Z, 0 \prec d$ such that $f(y) = f(y + d)$, where $d = B_p - B_k$ (7). But $y + d = G_p(y)$ and except for a number of points excluded due to the Gomory cuts, $G_p(\mathcal{C})$ is included in $\mathcal{C}$. Hence, $\exists y \in L_k \cap \mathcal{C} \cap \Pi_k$ such that $G_p(y) \in C$ (8). (7) and (8) imply that $\exists y \in L_k \cap \mathcal{C} \cap \Pi_k$ such that $f(y) = f(G_p(y))$, $G_p(y) \in L_k \cap \mathcal{C}$ and $y \prec G_p(y)$ (9). But (9) contradicts Proposition 9, and therefore, the assumption we started from is false.

Corollary 2 *Let C' be the Lexicographically Maximal Preimage domain of the problem given by (6.28). Then, $C' = (\bigcup_{j=1}^{m} \Pi_j) \cap \mathcal{C}$.*

Example 1: Let us see how the previous propositions, theorem and corollary apply to the example given in Figure 6.10.

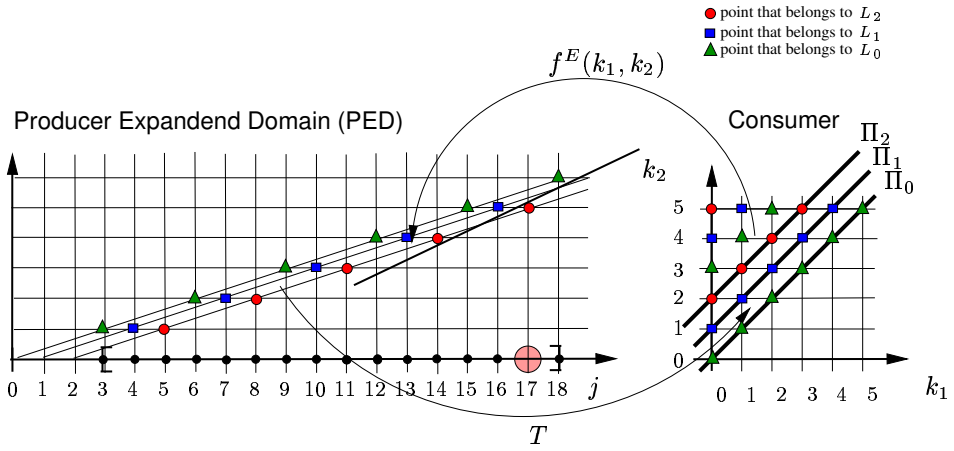


Figure 6.11: Lattice functions

As shown in Subsection 6.3.2, to find the lexicographically maximal preimage domain we first had to solve the following ILP problem:

$$\begin{aligned} \textbf{subject to:} \quad & 0 \leq k_1 \leq N, \\ & k_1 \leq k_2 \leq N, \\ & \mathrm{j} = 2k_1 + k_2 + 3, \\ \textbf{objective:} \quad & max_{\textbf{lex}}\{(k_1, k_2)\}. \end{aligned} \tag{6.30}$$

The solution to this problem is represented by the following multi-cast expression with only one non-empty branch:

```
1. if (0 <= j-3),
2.     if(0 <= -j+3*N+3),
3.       _d18 = div(2*j,6),
4.           if(0 <= -j+N+2*_d18+1),
5.               (k1,k2)=(_d18-1,j-2*_d18-1);
```

As you can see there is only one domain where the preimage is defined. Thus, we are under the restriction that there is only one validity domain for the preimage function. Hence, in order to obtain the lexicographically maximal preimage domain we can apply Corollary 2. But first let us investigate how Proposition 10 applies in this case. Recall from the example on page 88 that the explicit dependency function is:

$$f^E(k_1, k_2) = \begin{cases} (2 * k_1 + k_2 + 3, \frac{2*k_1+k_2+3}{3}) + (0, 0) & \text{if } (k_1, k_2) \in L_0^{\mathcal{C}}, \\ (2 * k_1 + k_2 + 3, \frac{2*k_1+k_2+3}{3}) + (0, \frac{-1}{3}) & \text{if } (k_1, k_2) \in L_1^{\mathcal{C}}, \\ (2 * k_1 + k_2 + 3, \frac{2*k_1+k_2+3}{3}) + (0, \frac{-2}{3}) & \text{if } (k_1, k_2) \in L_2^{\mathcal{C}}, \end{cases}$$

and therefore, there are three lattice functions: G_0 corresponding to lattice L_0, G_1 to lattice L_1, and respectively, G_2 to lattice L_2:

$$\begin{aligned}
G_0(k_1,k_2) = T \circ f_0^E(k_1,k_2) = &\quad (\tfrac{2*k_1+k_2}{3}, \tfrac{2*k_1+k_2}{3}) &&\text{if } (k_1,k_2) \in L_0^{\mathcal{C}},\\
G_1(k_1,k_2) = T \circ f_1^E(k_1,k_2) = &\quad (\tfrac{2*k_1+k_2-1}{3}, \tfrac{2*k_1+k_2+2}{3}) &&\text{if } (k_1,k_2) \in L_1^{\mathcal{C}},\\
G_2(k_1,k_2) = T \circ f_2^E(k_1,k_2) = &\quad (\tfrac{2*k_1+k_2-2}{3}, \tfrac{2*k_1+k_2+4}{3}) &&\text{if } (k_1,k_2) \in L_2^{\mathcal{C}}.
\end{aligned}$$

We can write now in a comprehensive way the three functions as:

$$G(k_1,k_2) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} + \begin{cases} \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \text{if } (k_1,k_2) \in L_0^{\mathcal{C}}, \\ \\ \begin{bmatrix} \frac{-1}{3} \\ \frac{2}{3} \end{bmatrix} & \text{if } (k_1,k_2) \in L_1^{\mathcal{C}}, \\ \\ \begin{bmatrix} \frac{-2}{3} \\ \frac{-4}{3} \end{bmatrix} & \text{if } (k_1,k_2) \in L_2^{\mathcal{C}}. \end{cases}$$

According to the above relation: $A = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$ such that $A^2 = A$.

Also observe that $B_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$,$B_1 = \begin{bmatrix} \frac{-1}{3} \\ \frac{2}{3} \end{bmatrix}$ and $B_2 = \begin{bmatrix} \frac{-2}{3} \\ \frac{-4}{3} \end{bmatrix}$, such that $AB_i = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \forall i = 0,1,2.$

Furthermore, as shown in the right part of Figure 6.11, by applying Corollary 2 we obtain the lexicographical maximal preimage domain as the intersection of the consumer domain with the union of the following planes:

$$\begin{aligned}
\Pi_0 : (k_1,k_2) = &\quad (\tfrac{2*k_1+k_2}{3}, \tfrac{2*k_1+k_2}{3}) &&\text{if } (k_1,k_2) \in L_0^{\mathcal{C}},\\
\Pi_1 : (k_1,k_2) = &\quad (\tfrac{2*k_1+k_2-1}{3}, \tfrac{2*k_1+k_2+2}{3}) &&\text{if } (k_1,k_2) \in L_1^{\mathcal{C}},\\
\Pi_2 : (k_1,k_2) = &\quad (\tfrac{2*k_1+k_2-2}{3}, \tfrac{2*k_1+k_2+4}{3}) &&\text{if } (k_1,k_2) \in L_2^{\mathcal{C}},
\end{aligned}$$

which can further be simplified as:

$$\begin{aligned}
&\Pi_0 : k_1 - k_2 = 0,\\
&\Pi_1 : k_1 - k_2 + 1 = 0,\\
&\Pi_2 : k_1 - k_2 + 2 = 0.
\end{aligned}$$

□

Our results have been obtained under the assumption that the problem has a single solution, thus is defined on a single domain. Next we analyze the case of multiple solutions and how the LMP domains corresponding the each of the solution can be calculated using Corollary 2.

Example 2: Assume the 2 dimensional consumer domain shown in Figure 6.12 specified as follows:

$$C = \{(x,y) \in Z^2 | 4y \geq -9x + 49 \wedge 5y \geq 2x - 3 \wedge y \leq -6x + 57 \wedge 6y \leq -x + 62\},$$

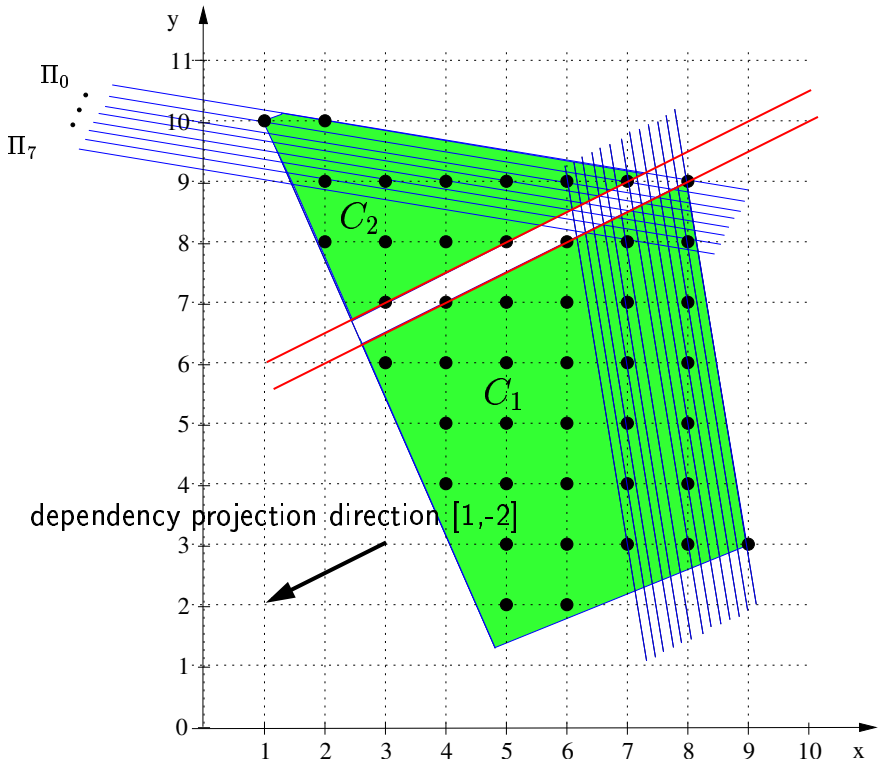


Figure 6.12: The LMP domains for a multi-domain solution tree obtained as union of planes intersected with the corresponding partition of the consumer domain. In this case the consumer is partitioned into two separated domains delimited by (red) lines.

and consider the following function begin the P/C dependency function:

$$f(x, y) = -x + 2 * y + 20.$$

We formulate the ILP problem that allows to derive the producer domain. Nevertheless this is out of the scope of this discussion so we don't present it. However, we are interested in the solution of this problem which is represented by the next multi-cast tree:

```
IfStatement: j-17
    IfStatement: -j+30
        IndexStatement: _d9 = div(12*j+4,13)
           IndexStatement: _d10 = div(_d9,2)
             a(-j+2*_d10+10,_d10-5)
    ElseStatement
       IfStatement: -25*j+986
           IndexStatement: _d11 = div(j+2,4)
              IndexStatement: _d12 = div(_d11,2)
                IfStatement: -9*j+22*_d12+241
```

We now now deduce the LMP planes for the second solution (in a similar way can be derived the planes corresponding to the first solution). Corresponding to the two solutions, the consumer domain

C is decomposed into two different sub-domains:

$$C_1 = \{(x,y) \in C \mid 17 \leq -x + 2y + 20 \leq 30\},$$
$$C_2 = \{(x,y) \in C \mid 31 \leq -x + 2y + 20\}.$$

For the second domain, the explicit dependency function is:

$$f^E(x,y) = \begin{cases} (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0,0) & \text{if } (-x+2*y+22)\%8 = 0, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-1}{8}) & \text{if } (-x+2*y+22)\%8 = 1, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-2}{8}) & \text{if } (-x+2*y+22)\%8 = 2, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-3}{8}) & \text{if } (-x+2*y+22)\%8 = 3, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-4}{8}) & \text{if } (-x+2*y+22)\%8 = 4, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-5}{8}) & \text{if } (-x+2*y+22)\%8 = 5, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-6}{8}) & \text{if } (-x+2*y+22)\%8 = 6, \\ (-x+2*y+20, \frac{-x+2*y+22}{8}) + (0, \frac{-7}{8}) & \text{if } (-x+2*y+22)\%8 = 7, \end{cases}$$

and therefore, there are eight G_i functions:

$$\begin{aligned}
G_0(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (0,0) && \text{if } (-x+2*y+22)\%8 = 0, \\
G_1(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-2}{8}, \tfrac{-1}{8}) && \text{if } (-x+2*y+22)\%8 = 1, \\
G_2(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-4}{8}, \tfrac{-2}{8}) && \text{if } (-x+2*y+22)\%8 = 2, \\
G_3(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-6}{8}, \tfrac{-3}{8}) && \text{if } (-x+2*y+22)\%8 = 3, \\
G_4(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-8}{8}, \tfrac{-4}{8}) && \text{if } (-x+2*y+22)\%8 = 4, \\
G_5(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-10}{8}, \tfrac{-5}{8}) && \text{if } (-x+2*y+22)\%8 = 5, \\
G_6(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-12}{8}, \tfrac{-6}{8}) && \text{if } (-x+2*y+22)\%8 = 6, \\
G_7(k_1,k_2) &= (\tfrac{3*x-6*y+62}{4}, \tfrac{-x+2*y+62}{4}) + (\tfrac{-14}{8}, \tfrac{-7}{8}) && \text{if } (-x+2*y+22)\%8 = 7.
\end{aligned}$$

As shown in Figure 6.12, by applying Corollary 2 we obtain the lexicographically maximal preimage domain as the intersection of C_2 with the union of the following eight planes:

$$\Pi_0 : 6*y + x = 62, \quad \Pi_1 : 6*y + x = 61,$$
$$\Pi_2 : 6*y + x = 60, \quad \Pi_3 : 6*y + x = 59,$$
$$\Pi_4 : 6*y + x = 58, \quad \Pi_5 : 6*y + x = 57,$$
$$\Pi_6 : 6*y + x = 56, \quad \Pi_7 : 6*y + x = 55.$$

□

6.3.3 Communication synthesis / Address generation

We now turn to the address generation for the memory used to store data in each of the network communication channels. As explained in Section 6.2, we have for each channel a *write* and a *read* function that are used by the producer process to write data to the communication memory, and by

the consumer process to read data from the communication memory, respectively. These functions do not necessarily have to be identical to those used for addressing the memory in the sequential program. However, the current compiler implementation does not allow to address out of order channels based on different addressing functions than those used in the sequential program. In general, we would like those addressing functions to be as simple as possible to allow for fast computation without obstructing the flow of the data through the network, in particular when targeting a hardware platform with limited resources. For this purpose, we take into consideration the classification of the communication patterns as obtained in the Communication Model Selection step.

The communication structure for the different communication types increases in complexity from IOM- to OOM+. The implementations of IOM- and IOM+ channels are closely related, except that for IOM+ additional control is needed to know when to read new data from the FIFO. The implementations of OOM- and OOM+ channels make use of a random or a content addressable memory, as data needs to be reordered at run-time. OOM+ requires additional control to know when to release a certain location in memory. Of the four types identified, OOM+ is the most expensive. It is also the most generic communication type since it subsumes all three other types (IOM-, IOM+, and OOM-). Consequently, the OOM+ communication structure can be used to implement the other three communication types as well. In [92] we have presented a number of alternative realizations of this type of communication out of which two types are described in this thesis.

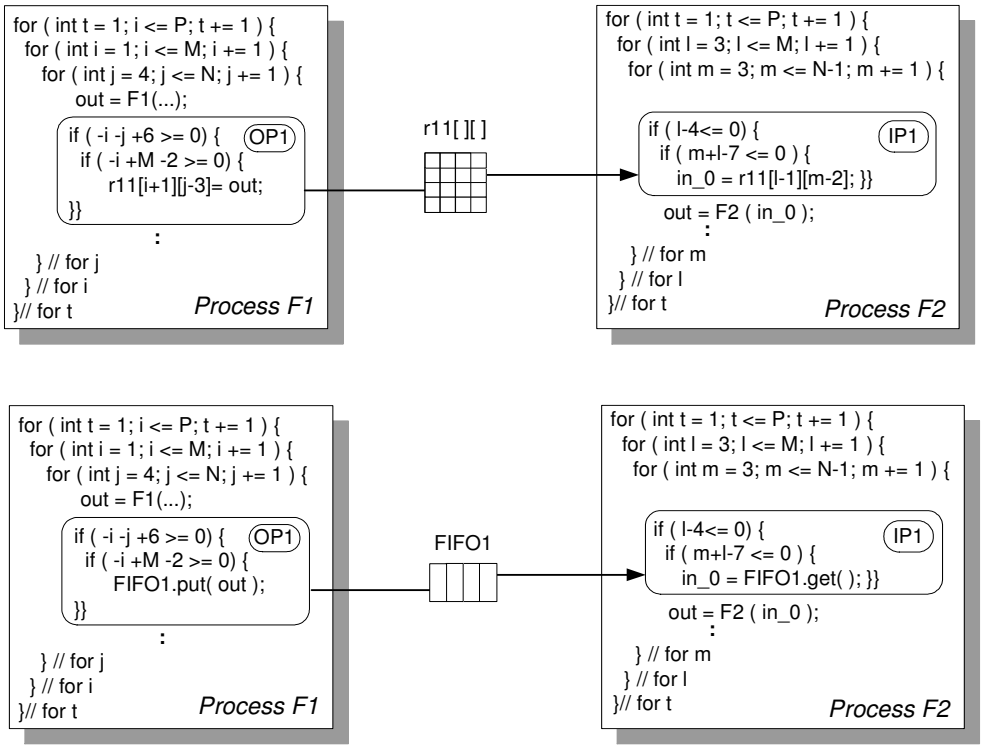


Figure 6.13: Address generation for an IOM- P/C pair

Buffer sizes

Another important issue concerns the memory size which ensures a deadlock free network. For this purpose we allocate for each of the communication channels the same amount of memory as used for storing data in the sequential program. Another possibility is to fix the program loop boundaries and then to simulate the final network using e.g., the Ptolemy simulator. As a result we get a fixed number of memory locations for each communication channel. More work regarding the minimal buffer sizes is further given in Section 8.2.2.

Address generation for an IOM- P/C pair

In the case of an IOM- P/C pair, the order of writing data is the same as the order of reading data, and each written token is read only once. Instead of computing the write and the read functions as given in the sequential program, we can generate them based on two counters that are incremented at each activation of the communication channel. Thus, such communication can be synthesized as a FIFO buffer that is accessed using a `get` and a `put` primitive. Addressing the memory in a FIFO fashion using a get/put primitive is very beneficial from the flow of data point of view, i.e., the addressing of the memory will not obstruct the pure computation that is executed by the network.

Running example

For the running example, the producer/consumer pair of processes $F1/F2$ is transformed as shown in Figure 6.13. Observe how the original addressing primitives based on a multidimensional address have been replaced by a `FIFO.put` respectively `FIFO.get` primitive.

Address generation for an IOM+

In the case of an IOM+ P/C pair the order of writing in the channel is the same as the order of reading from the channel but some of the written tokens are read more than once. As in the IOM- case, the read and write addressing functions present in the sequential code are replaced by two counters that are incremented at each activation of the communication channel. Thus, such communication can be synthesized as a FIFO buffer that is accessed using a *get* and a *put* primitive. However, as a token will be read more than once, we need to keep track of the life-time of a token. This is realized by guarding the get primitive with control representing the lexicographical minimal preimage domain. As shown in Section 6.3.2, the lexicographically minimal preimage LmP domain is a subset of the consumer domain that consist of all the iterations at which new data is consumed for the first time. Therefore, at each iteration in the LmP data is read from the FIFO and stored in an additional register from where later on is consumed consecutively for a number of times.

Example IOM+: The LmP domain and the associated linearization the second example from Section 6.2.1 is shown in Figure 6.14. As shown there, for each consumer iteration in the LmP data is read from the FIFO and stored in the local variable `token`, from where it is consumed by a number of consecutive process firings.

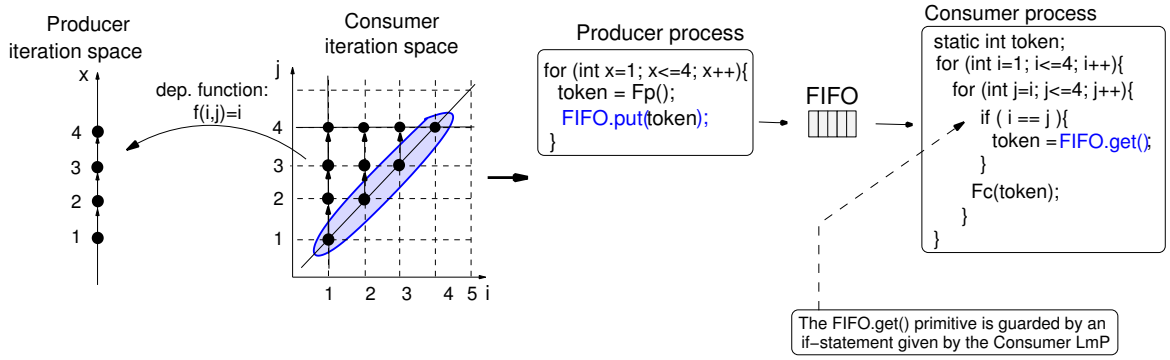


Figure 6.14: Communication for an IOM+ P/C pair

Running example

For the running example, the F1/F3 P/C pair is transformed as shown in Figure 6.15. For to the consumer domain $F4$, two lexicographically minimal preimages exist (i.e., LmP_1, and LmP_2). Thus only at an iteration that belongs to those two domains, data is read from the FIFO and stored in the local register; elsewhere the data is re-read from the local register. In Figure 6.15 this register is represented by the statically declared variable `static int in_0`.

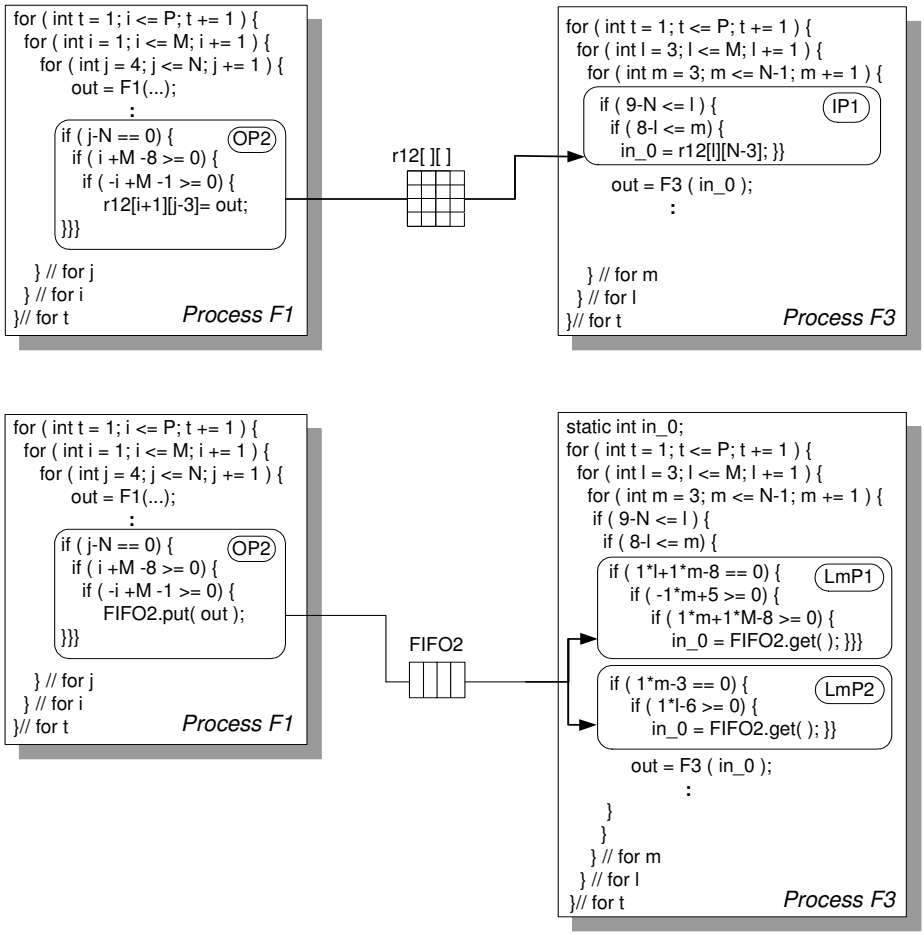


Figure 6.15: Address generation for an IOM+ P/C pair

Address generation for an OOM-

The communication structure used for synthesizing OOM- channels is more complex than the one used for IOM- channels. OOM- requires that the communication memory is random accessible using a read address (at the consumer side) and a write address (at the producer side). We can generate those addresses based on the $write$ and $read$ functions associated to each channel. Each memory location is augmented with a validity bit. However, those functions can only be used for a software representation of the final network specification. When considering a hardware target, the multidimensional arrays are not directly synthesizable. To solve this problem we have to derive $write_l$ and $read_l$ functions that take values in a unidimensional address space. Those functions are obtained taking into account the producer domain and the $write$ function. Because multiplicity is not involved, each time a consumer process reads data from a memory location, the validity bit associated with that location is set to zero allowing the producer process to overwrite that location.

In [92], we have investigated four different realizations. In this thesis we present only two of them as they are supported by the VHDL back-end [113, 109]. To introduce the two realizations, we use as an example the following P/C pair:

Producer:

```
for (int i=1;i<=N+2;i++){
   for (int j=1;j<= N;i++){
      if (2*j >= i+6){
          a[i,j] = Fp();
      }
   }
}
```

Consumer:

```
for (int y=4;y<= N;y++){
   for (int x=1;x<= N+2;x++){
      if (x <= 2*y-6){
          Fc(a[x,y]);
      }
   }
}
```

The graphical representation of the two iteration spaces is shown in the top part of Figure 6.16. For the sake of simplicity we have chosen at the producer side for memory allocation purposes the identity function $write(i,j) = (i,j)$. Hence, the memory space is identical to the producer iteration space.

Linear realization

The Linear realization makes use of a RAM memory and is based on the classical linearization of an n-dimensional array into a 1-dimensional array as given in Chapter 8 of [1]. The classical linearization shows that a rectangle shape can be addressed using linear read/write functions. To derive the $write_l$ function we first use PolyLib [105] to map the producer OPD through the $write$ function and get the memory space $MemSpace = write(OPD)$. Then we bound the memory space to a rectangular that includes the memory space: $MemSpace'$. Normally we would like the volume of $MemSpace'$ to be as close to the volume of $MemSpace$. Due to the bounding, a certain amount of memory locations are wasted (see the locations containing `NULL` in Figure 6.16). The smaller this waste is the better. Bounding a memory space is a memory reuse problem. There are several papers dealing with the analysis of memory reuse in programs that can be represented using the polyhedral model. In [98], an approach is presented for a fixed linearization of the memory array. In [104, 106, 76], a constructive approach is given in the context of the single assignment language ALPHA, for finding the maximum lifetime of an array, based on which an optimal memory projection can be derived. In [29], an approach is presented to compute the bounding box for the elements that are simultaneously in use. The size

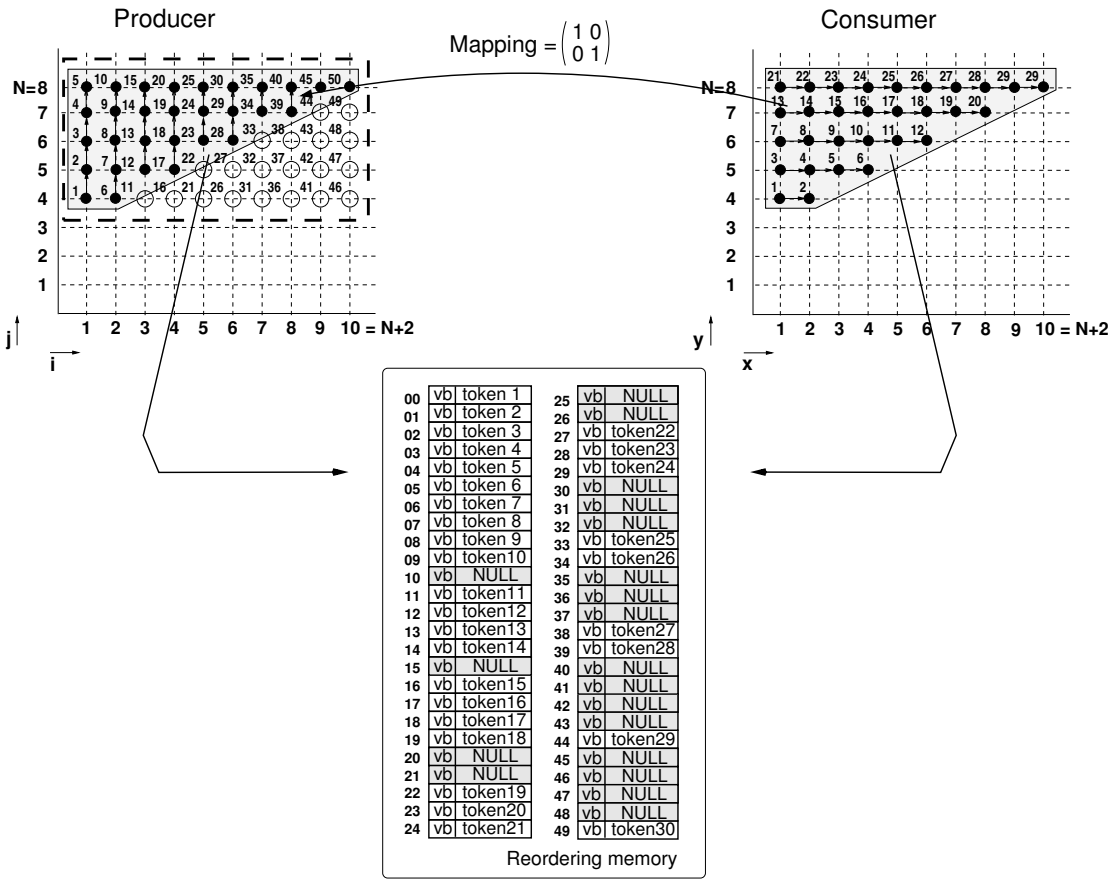


Figure 6.16: The Linear realization

of the original array is reduced to the bounding box and accessed using modulo operations, giving an optimal memory usage. More recently, a global view over these techniques has been given in [27].

Based on integer linear programming [37], our linear realization derives $MemSpace'$ as a rectangular bounding box with parametric vertices [110]. For our running example this set is depicted in Figure 6.16, where the $write_l$ function is as follows:

$$write_l(i,j) = (N-3) * (i-1) + j - 4. \tag{6.31}$$

The $read_l$ function is obtained by composing the write function with the P/C dependency function $M(x,y)$:

$$\begin{aligned} read_l(x,y) &= write_l(i,j) \circ M(x,y), \\ &= (N-3) * (x-1) + y - 4. \end{aligned} \tag{6.32}$$

In the context of the FPGA target architecture, the advantage of this realization, is that the functions used to address the reordering memory are always linear expressions depending on the coordinates of the consumer or consumer ISs [112]. A disadvantage is that there are extra iteration points in the linearization bounding box resulting in empty memory locations (represented by the

"Nulls" in the memory in Figure 6.16). In this example, the memory size is equal to the volume of the linearization box, i.e., $V(MemSpace') = (N-3)*(N+2)$, but only half of this space is actually used.

CAM realization

To overcome the Linear realization waste of memory, we use a Content Addressable Memory (CAM) as reordering memory. In a CAM, a *key* is used instead of an address to access the content of the memory, and data is stored at the first free memory location. In this way, we get rid of all the empty memory locations at the expense of a more complex addressing mechanism. The entry used in the CAM realization is given in Figure 6.17.

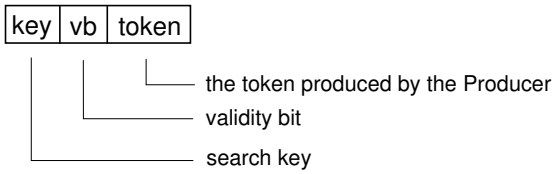


Figure 6.17: The CAM entry

Each memory entry consists of a key, a validity bit and a token. To each token produced at the producer OPD a *unique* key is associated. This key is used instead of an address for writing to the memory. The generation of the key can be done in different ways. One possibility is to write data into the memory under a key equal to the $write_l$ function. Similarly, at the consumer side data is consumed using a key given by the $read_l$ function.

For a given iteration point (x, y), at the consumer side, the *read* function determines a key given by the $read$ function. If the key exists and $vb = 1$, then the associated token is retrieved for consumption from the memory. In case no match can be found, or in case the match was found but the validity bit is zero, then the consumer blocks, waiting for a valid token to be stored. Thus, if the key exists and the validity bit is set to 1, then the token associated with the key is retrieved from the CAM and passed to the consumer execution function, followed by the invalidation of the address by reseting the validity bit $vb = 0$.

Address generation for an OOM+

To realize the communication in case of an OOM+ channel, we can use the two OOM- realizations presented so far. However, due to multiplicity, we have to keep track of the last consumption of a token in order to reset the validity bit. For this purpose, we make use of the lexicographically maximal preimage domain as obtained in Section 6.3.2. When the consumer iterates inside the LMP domain, the life-time of the consumed token has come to its end, and the memory location is released by setting the corresponding validity bit to zero.

Example OOM+: Here we present the network derived from a program that is a slightly modified version of the one given in Example 2 in Section 6.3.2. To introduce multiplicity we have added an extra k loop:

```
for (int k=0; k<=T; k++){
  for (int j=0; j<=3*N+3; j++){
      a[j] = Fp();          // Producer
  }
  for (int k1=0; k<=N; k++){
    for (int k2 = k1; k2<=N; k2++){
       Fc( a[2*k1+k2+3] ); // Consumer
    }
  }
}
```

As a result, the $write(k, j) = j$ function is no longer injective such that multiple producer iterations assign data to the same location. As a consequence, the P/C pair becomes out of order with multiplicity. Because the producer writes in a unidimensional memory array, there is nothing to be linearized. Thus, the example concentrates on the memory reuse. After applying the procedure described in Section 6.2.1, we get a LMP domain as follows:

$$C = \{(k_1, k_2) \in Z^2 | 0 \leq k \leq T \wedge 0 \leq -k_1 + k_2 \wedge k_2 \leq N \wedge 0 \leq 2k_1 - 2k_2 + 5 \wedge 0 \leq 2k_1 + k_2\}.$$

As you can see in the final representation in Figure 6.18, the LMP domain is represented by a single if-statement: $if(0 <= 2 * k1 - 2 * k2 + 5)$. This is due to the simplification of the LMP in the context of the consumer domain.

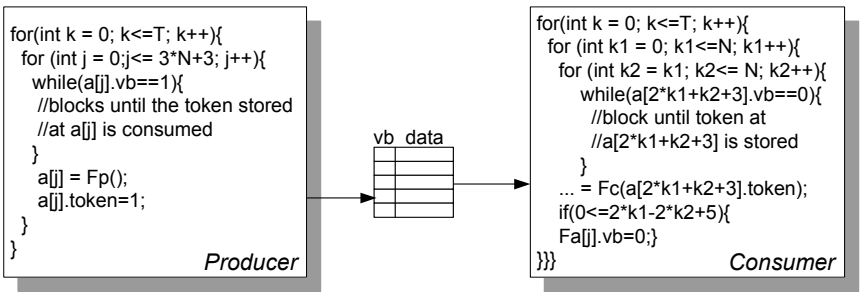


Figure 6.18: The final network implementation for OOM+

6.4 Running example

Let us analyze the final software network implementation for the running example. The network is visualized in Figure 6.2. In this example, as in most real life cases, we are confronted with communication of types IOM+ and IOM-. For PC_1, PC_2, PC_3, we replace the static arrays $r11$, $r21$ and $r22$ with a FIFO buffer. Observe that the absolute addressing performed on the arrays is now replaced by a relative addressing using Put and Get primitives. For PC_4, we replace the static array $r12$ by a FIFO buffer, taking into account the life-time of the tokens due to multiplicity. To determine when a process can read from FIFO2 we make use of the LmP. We map the domain represented by OPD_2 through the corresponding solution functions. Hence, we map the domain D_1 through the affine mapping $(t_p, i_p+1, -i_p+7)$, and we get $LmP_1 = \{(t, l, m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l+m = 8, n = 3, m \leq 5\}$. Similarly, we map domain D_2 through the affine mapping $(t_p, -i_p + 1, 3, 3)$, and we get domain

$LmP_2 = \{(t,l,m) \in Z^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 3 \leq m \leq M, m = 3, 6 \leq l \leq 9-N\}$. Once LmP_1 and LmP_2 have been derived, we can simplify them in context of the correspondent process IPD by removing constraints in common with the constraints describing the IPD.

The pseudo code for the PN is shown in Figure 6.19. It shows the way the four processes are implemented. It also shows how the IPDs and OPDs derived in the various steps, are transformed into if-statements using linear expressions. In case of Process $F3$, we need to implement the constraints that take care of the life-time of tokens into account. The simplified LmP_1 and LmP_2 are converted to if-statements and inserted into the IPDs of Process $F3$. If conditions LmP_1 or LmP_2 hold, a token is read from the FIFO and is reused as many times as needed before the $LmPs$ indicate that the next token needs to be read.

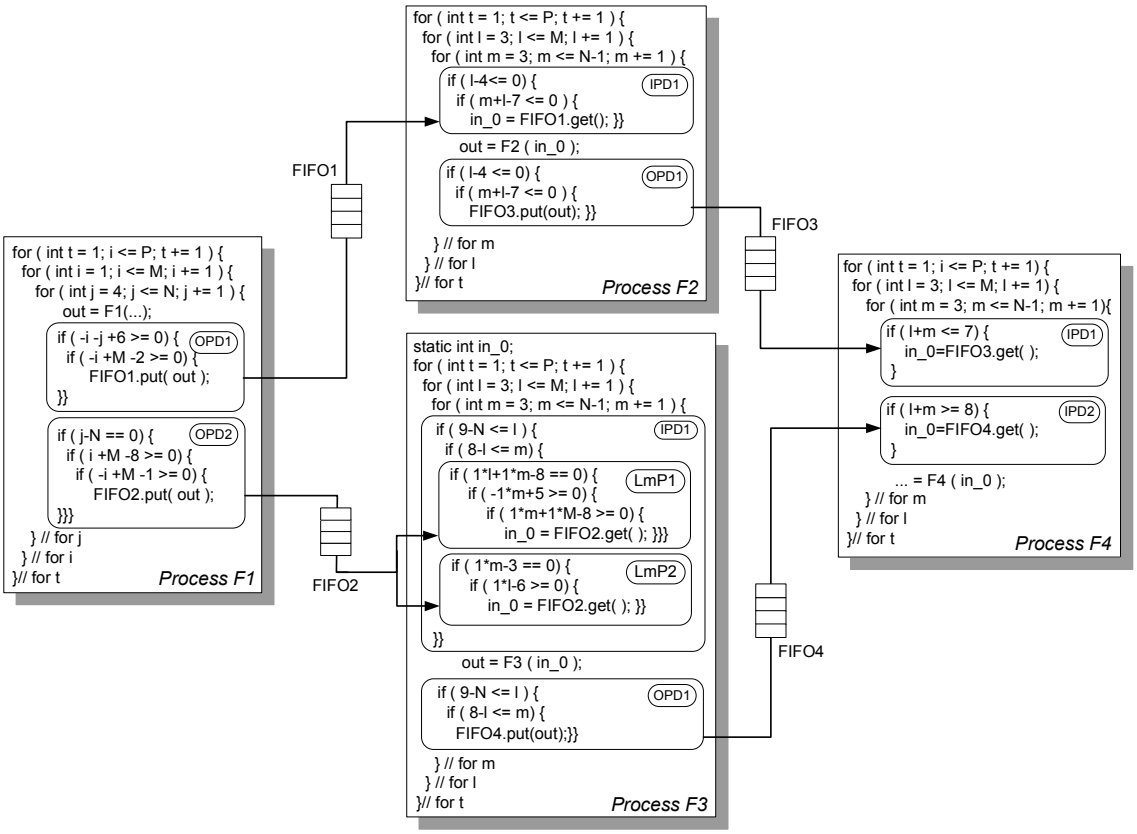


Figure 6.19: The final network implementation

Chapter 7

Network Transformations

As presented so far, our compilation approach derives a unique process network for a given input program. However, it is possible that the network does not fit the components of the target platform or that the overall achieved performance obtained as a result of the mapping is not satisfactory from the point of view of data throughput and/or memory usage. In this chapter we address this problem by introducing two network transformations which make it possible to derive alternative process networks from one and the same input program. These transformations manipulate the process network structure obtained after performing the Process Network Synthesis step described in the previous chapters.

Below we briefly summarize the features of such a network structure. In general a PN is represented as a directed graph in which the nodes are processes denoted as F_k and the edges are communication channels denoted as Ch_l. The structure of a process F_k is characterized by:

- an assignment statement that is the core of the computation in the process. To each assignment corresponds a function call F_k which is free of side effects. This means that *all the data dependencies within the original application are exposed via the communication channels*. Once the process starts executing, data is read from some of the incoming channels, passed as input arguments to the process function whose output is written to some of the outgoing channels.
- a local loop schedule iterating over an iteration space (IS_{F_k}). In general the iteration space is represented by a linearly bounded lattice (LBL). However, without loss of generality (based on the procedure presented in Section 4.1.1) we consider the iteration space to be a parametric polytope.
- a number of input ports (IPs), where an input port IP_i represents a parametric polytope which is a subset of the process iteration space $IP_i \subset IS_{F_k}$ containing only iterations at which one of the arguments of a process function is read from the same incoming channel. At each process firing, some of the input ports are delivering data to be processed by the internal function. Each input port contains a `channel.read(adr)` statement where `adr` is the address from which data has to be consumed. In case of a FIFO channel the read address is implicitly the first-out location. Hence, the read primitive becomes `channel.read()`.
- a number of output ports (OPs), where an output port OP_i is an LBL subset of the process iteration space $OP_i \subset IS_{F_k}$ containing only iterations at which one of the outcomes of a process

function is put into the same outgoing channel. At each process firing, some of the output ports are active, i.e., the data is written to some of the outgoing channels. Each output port contains a `channel.write(adr,token)` statement where adr is the location at which the token has to be stored. In case of a FIFO channel the write address is implicitly the first-in memory location. Hence, the write primitive becomes `channel.write(token)`.

In this PN structure, a channel Ch_l uniquely connects an output port OP_i to an input port IP_j. Each such channel is characterized by two functions:

- **data dependency (f) function:** a mapping function $f_{Ch_l} : IP_j \rightarrow OP_i$ obtained after performing the Consumption Restructuring step. This function is the affine dependency between the iterations of IP_j and those of OP_i. Recall that even if the communication channel Ch_l has been synthesized as a FIFO channel, the function f_{Ch_l} is not necessarily injective.
- **lexicographic minimal/Maximal preimage (lmp) function:** After performing the Process Network Synthesis step we obtain, for each channel Ch_l, the lexicographic minimal and Maximal functions: $lmp_{Ch_l}/lMp_{Ch_l} : OP_i \rightarrow IP_j$. The lmp_{Ch_l}/lMp_{Ch_l} express the dependency between the iterations of OP_i and those of IP_j at which the first/last consumption of a produced token takes place.

Notice that the output and the input ports of a channel may belong to the same process. Such channel is referred to as an *inner channel*.

In the sections to follow we first present a transformation used to increase the parallelism at the network level by *splitting* network processes. The second transformation is used to *refine* the network communication structure. This transformation is a generalization of the Communication Model Selection, i.e., it analyzes the opportunity of merging a number of different FIFO channels without introducing reordering or extra control overhead. Both transformations are worked out on eloquent examples. From the compiler implementation point of view, the first transformation was only prototyped. However, the outcome and effectiveness of the first transformation has been investigated by performing two source code transformations (i.e., skewing and unfolding) at the level of the compiler input program [84, 82, 83]. The second transformation is fully implemented allowing us to present its effectiveness.

7.1 Process Splitting

In this section, we address the problem of increasing parallelism in a process network obtained after performing the first four compilation steps. The increase of network parallelism is achieved by a transformation called *Process Splitting*. This transformation exploits data parallelism at the network level and locally at the process level. By using the Process Splitting transformation an arbitrary process can be split in a number of autonomously running processes. The splitting transformation presented here covers the set of three source code transformations introduced in [83]. Our approach exhibits the following characteristics:

- **scalability**, i.e., the approach is not affected by the increment of the splitting factor (i.e., the number of times a process is split). For the approach presented in [83], this induces an increase of the number and the dimension of integer linear programs that have to be solved. For some real life applications this leads to the incapability of the ILP solver to find solutions during some

of the compilation steps, although for the same unsplit[1] program the compilation performs correctly.

- **flexibility**, i.e., the user has the freedom to operate on an arbitrary process of a network. Because it operates on the loop structure of the input program, the approach from [83] does not allow the user to specify a certain process that will be split, but rather a loop nest level at which all the embedded assignment statements will result in split processes. For example if the original code contains two assignment statements located at the same nesting level, then the choice to split one of the processes (that results from one of the statement) impacts the splitting of the second one as well.
- **legality**, i.e., our transformation is proved to be correct under the assumption that the process function calls are free of side effects, i.e., all the dependencies between different calls to the same function are only those exposed via the input and output function arguments; otherwise the functions have to be preprocessed and brought to such a form. Compared to our transformation, the skewing transformation described in [83] does not guarantee the preservation of the original application functionality. In order to be applied correctly the user has to spend a considerable amount of time analyzing the dependence graph of the original application.

As shown in Figure 7.1, we position *Process Splitting* after the Process Network Synthesis step. The key observation that makes our approach successful is that *since the data-dependence analysis has been done for the input program, the splitting of a process results in a regrouping of the original program iteration space without the overhead incurred by solving false dependencies as present in the transformed (unfolded[83]) input program.*

7.1.1 Solution

In this sub-section we present the process splitting transformation. Section 7.1.2 provides an example. The transformation involves three main steps, and before being applied the user has to specify:

- a process (or a set of processes) that has to be split,
- a splitting factor sf representing how many times the process has to be split, and
- a partitioning function $p : IS \rightarrow Z$ which in combination with the splitting factor associates with each iteration point x of the original process iteration space IS a number j representing the split process to which the point belongs after the splitting transformation. Based on the partitioning function, the splitting is modeled as follows:

$$\text{if } \ (\ p(x)\%sf == j \ ) \ \text{ then after splitting } \ x \in P^j, \tag{7.1}$$

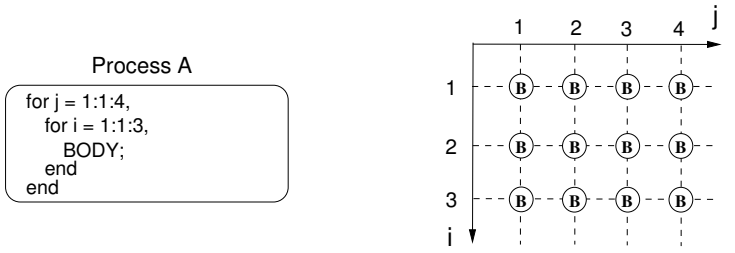
where $j = 0, \ldots, n-1$ represents the number of the process to which x belongs after splitting.

Next we present the steps in the splitting transformation followed by a discussion on the correctness of the transformation:

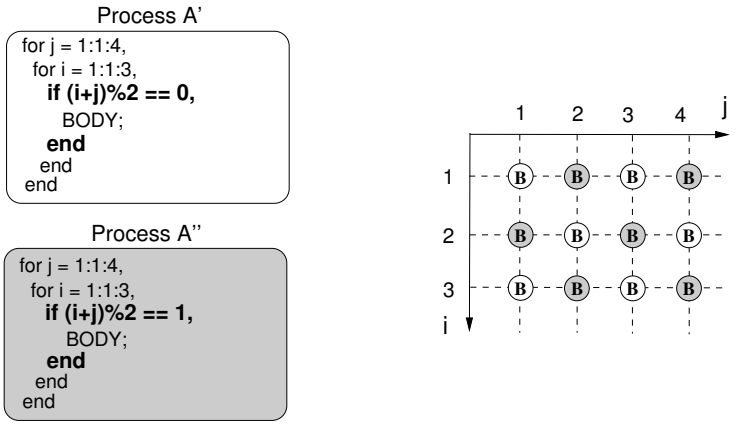
Step 1 (Process duplication with modulo selection): As a result of the first step a process is split based on code restructuring by duplicating sf-times the original process control structure, keeping

[1] not unfolded - with the terminology used in [83]

the same execution function. Figure 7.1 shows the effect of splitting a process loop nest structure by a factor of 2. For the sake of clarity, we mask the internals of the process control structure (i.e., input/output ports and process function) with a BODY statement (B). The internal control structures of the duplicated processes are rearranged by inserting adequate modulo conditional expressions. The



a) Original process loop body structure with it's iteration space



b) The processes loop structures after a splitting with factor 2 together with the new allocation of internal process structure

Figure 7.1: Process splitting using modulo conditional expressions

conditional expressions used for splitting are shown with bold characters in Figure 7.1. Observe that the splitting factor is $sf = 2$ and the partitioning function is $p(i,j) = i + j$. In this case the points (i,j) for which $p(i,j)\%2 = 0$ will be executed by the first process A', while the points for which $p(i,j)\%2 = 1$ will be executed by the second process A''.

Step 2 (Deriving the new IPD/OPD control structure): In the second step of the approach, we establish the control structure for all the processes affected by the first step. Basically we have to adapt the control only for the ports that are connected via channels to split processes.

The partitioning of the input and output port domains is done by taking into account the dependency function f and the lexicographically minimal preimage lmp [2]. This issue will be detailed later, within the correctness discussion. During the splitting of a process, the incoming and outgoing FIFO channels are duplicated as well: for each new process the names of the channels being annotated with the number of the process.

[2]Here the purpose of using the preimage function is only to find out whether for a produced token a process consumes that token at a certain firing.

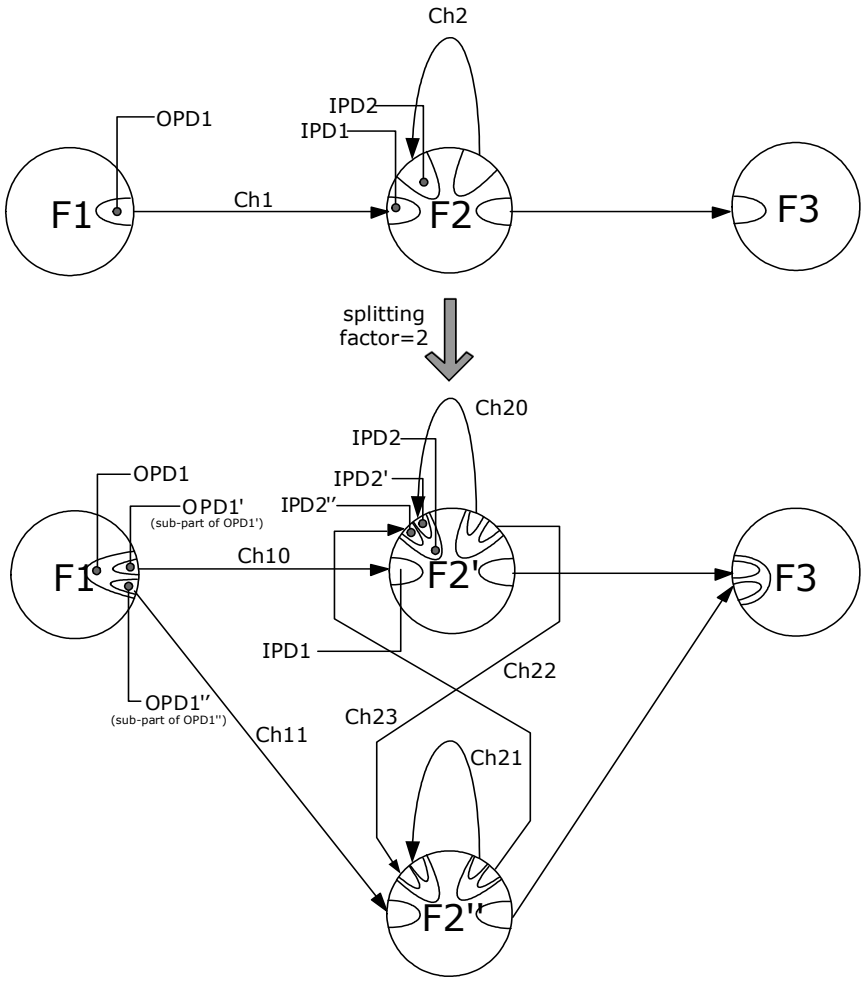


Figure 7.2: The topological aspects of splitting 2 times the process $F2$

The topological aspects of the splitting transformation are graphically depicted in Figure 7.2. We have chosen to split two times process $F2$, partitioning the output and input ports and connecting the communication channels adequately. The first thing to notice is that in process $F1$ the output port $OPD1$ is partitioned into two ports: $OPD1'$ and $OPD1''$, where $OPD1'$ consists of the iteration points at which data has to be communicated to process $F2'$ and $OPD1''$ consists of the iteration points at which data has to be communicated to process $F2''$. Notice that in case channel $Ch1$ has multiplicity then $OPD1'$ and $OPD1''$ are not necessarily disjoints sets.

Similarly for process $F2'$ the input port $IPD2$ is partitioned into two sub ports $IPD2'$ and $IPD2''$, where $IPD2'$ consists of the iteration points at which data has to be consumed by process $F2'$, and $IPD2''$ consists of the iteration points at which data has to be consumed by process $F2''$. Notice that independently of the multiplicity characteristics of $Ch2$, $IPD2'$ and $IPD2''$ represent disjoint sets. Channel $Ch1$ connecting process $F1$ with process $F2$ is split as well. For process $F2'$ the incoming channel is $Ch10$, and for process $F2''$ the incoming channel is $Ch11$. The same applies for the channel connecting process $F2$ with process $F3$. However, in case of an inner channel, this channel is replaced by (at most) four channels: an inner channel for each of the new obtained processes and two more

channels interconnecting them. In Figure 7.2, the inner channel $Ch2$ has been replaced by 2 inner channels $Ch20$ and $Ch21$, and two inter-process channels $Ch21$ and $Ch23$.

Step 3 (Control refinements): In the third step of the splitting transformation, the network is optimized using parametric integer linear programming. This optimization is useful because some of the generated port domains may be empty which leads to communication channels through which data will never be communicated. To check whether a port domain is empty, we make use of the Empty Domain Test that was introduced in Section 2.3.4. Based on this optimization, input and output ports together with their corresponding communication channels are removed.

Correctness discussion

In order to preserve the functionality of the original network, we have to partition IPDs and OPDs involved in splitting, i.e., ports that are part of the process which is being split, and ports connected via channels to this process. To be correct, the splitting has to respect the data-dependencies of the original network. As already introduced in Section 6.2.1 the original network (onto which the transformation is preformed) is functionally equivalent to two sequential program and exposes the following types of process dependencies:

- output dependencies between firings of the same process with respect to the writing of data into the same communication channel, and
- flow dependencies between producer/consumer firings.

The splitting transformation has to preserve those two types of dependencies. First we show how the output dependencies are preserved by default. Then we derive additional control conditions under which the flow dependencies are preserved as well.

Output dependencies: Let F_x be the process chosen to be split. Consider $F_x(ip_i)$ and $F_x(ip_j)$, the two firings of the process function executed at iterations ip_i and ip_j, respectively. We observe the following.

- **Remark 1:** if $F_x(ip_i)$ is fired before $F_x(ip_j)$ (i.e., $ip_i \prec ip_j$) and if as a result of the splitting they are regrouped into the same process $F_x^{'}$ then they preserve the same order of firings, namely $F_x^{'}(ip_i)$ is fired before $F_x^{'}(ip_j)$.
- **Remark 2:** if as a result of the splitting they appear into different processes: $F_x^{'}$ and $F_x^{''}$, then they write data at the same address but into different communication channels.

Based on Remark 1 and Remark 2, we now investigate how the output dependences appearing in the original network are affected by the splitting transformation. Formally, if $F_x(ip_i)$ and $F_x(ip_j)$ are two output dependent firings of process F_x, i.e., $F_x(ip_i)\ \delta^o\ F_x(ip_j)$ then in the split network:

- if the two iterations belong to the same process i.e., $ip_i, ip_j \in F_x^{\prime}$, then their output dependency is preserved due to Remark 1.
- if the two iterations are partitioned into two different processes i.e., $F_x^{'} \neq F_x^{''}$, then the dependency is removed because as stated by Remark 2 they write data into different channels, by Remark 2.

We have thus the following property.

Property 2 *The splitting transformation based on mutually exclusive conditional modulo-statements does not violate the output dependencies existing within the original network.*

Flow dependences: Property 2 simplifies our further analysis. Thus to preserve the network functionality, the extra control needed to partition the ports has to take only of respecting the flow dependencies into account. Suppose that a process F is split sf times: $F^j,\ 0 \leq j < sf$. To preserve the flow dependences in the resulting network configuration we need to:

- for an iteration point ip_x belonging to an IPD of a process that consumes data from the split process F, determine from which of the new processes data has to be read. Consider the network presented in Figure 7.2. When iterating inside $IPD2$, the process $F2'$ has to decide from which of the processes $F2'$ or $F2''$ data has to be consumed. Similarly when iterating inside $IPD3$, process $F3$ has to decide from which of the processes $F2'$ or $F2''$ it has to consume data.
- for a point ip_x belonging to an OPD of a process that sends data to the split process F, determine in the new network configuration to which of the new processes data has to be sent. Consider Figure 7.2: when firing an IP belonging to $OPD1$, process $F1$ has to decide to which of the processes $F2'$ or $F2''$ data has to be sent to.

These two problems can be solved by providing a compile time partitioning of the ports involved in the splitting transformation as visualized in Figure 7.2.

The first problem deals with partitioning of an input port domain IPD of a process that consumes data from process F which is being split. This partitioning can be established using modulo conditional expressions. If ip_x is an IP that consumes data from process F, then based on the dependency function f we certainly know that $ip = f(ip_x)$ is the corresponding iteration that produces that data. Based on the splitting function p, the iteration ip will be executed by one of the new processes.

$$\begin{array}{lll} \text{if} & (p(ip)\%sf == 0) & \text{then} \quad ip \in F^0, \\ \text{if} & (p(ip)\%sf == 1) & \text{then} \quad ip \in F^1, \\ : & & \\ \text{if} & (p(ip)\%sf == sf-1) & \text{then} \quad ip \in F^{sf-1}, \end{array}$$

where sf is the splitting factor. Therefore, the partitioning of IPD is based on the following conditions:

$$\begin{array}{lll} \text{if} & (p(f(ip_x))\%sf == 0) & \text{get data from process} \quad F^0. \\ \text{if} & (p(f(ip_x))\%sf == 1) & \text{get data from process} \quad F^1, \\ : & & \\ \text{if} & (p(f(ip_x))\%sf == sf-1) & \text{get data from process} \quad F^{sf-1}. \end{array}$$

Observe that these conditions are mutually exclusive as each consumption requests data produced by a unique producer process.

In case f is an injective function, the partitioning of the OPD can be done in a similar manner. In this case, the lmp function uniquely connects a production to a consumption ip. Hence, if ip_x is an iteration of a process that produces data consumed by process F, p is the partitioning function, and

sf is the splitting factor, then:

$$\begin{aligned}
&\text{if } (p(lmp^{-1}(ip_x))\%sf == 0) && \text{send data to process } F^0,\\
&\text{if } (p(lmp^{-1}(ip_x))\%sf == 1) && \text{send data to process } F^1,\\
&\vdots\\
&\text{if } (p(lmp^{-1}(ip_x))\%sf == sf-1) && \text{send data to process } F^{sf-1}.
\end{aligned}$$

However, if f is not injective, then a produced token may be consumed by several of the new processes obtained by splitting. This implies that we cannot split the OPD based on lmp. Instead, the Production Restructuring and the Life Time analysis part of the Code Generation have to be performed for each of the IPD/OPD pairs involved during the splitting transformation. This is done by taking into account the modulo conditional expressions used to describe the IPDs of these processes. Consider the IPD being an input port affected by the splitting function p and a splitting factor sf. Then, the Production Restructuring and the Life Time analysis will be performed for each of the new obtained ports $IPD^j, j = 1..sf$, where:

$$IPD^j = \{ip \in IPD \mid p(ip)\%sf = j\}. \tag{7.2}$$

By setting up for each of the IPDs an ILP system based on equation (4.1), we obtain the representation of each of the corresponding OPDs. Then, for each of the new P/C pairs, we obtain the lexicographic maximal or minimal preimages based on equation (6.8). See also the example in sub-section 7.1.3.

7.1.2 Example - splitting without multiplicity

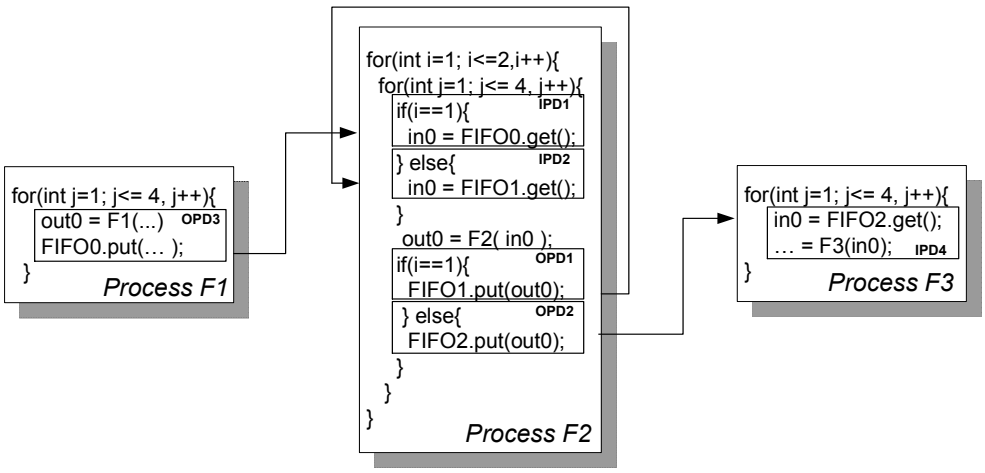


Figure 7.3: Starting network

Consider the network shown in Figure 7.3. This network consists of three processes and three communication channels. Each channel communicates data in order without multiplicity (IOM-). The code associated with each of the processes is shown as well, allowing to easily identify the input and output ports for each of the component processes. We choose to split process $F2$ by a factor of 2 based on the partitioning function $p(i, j) = j$. This means that the even iterations from the second

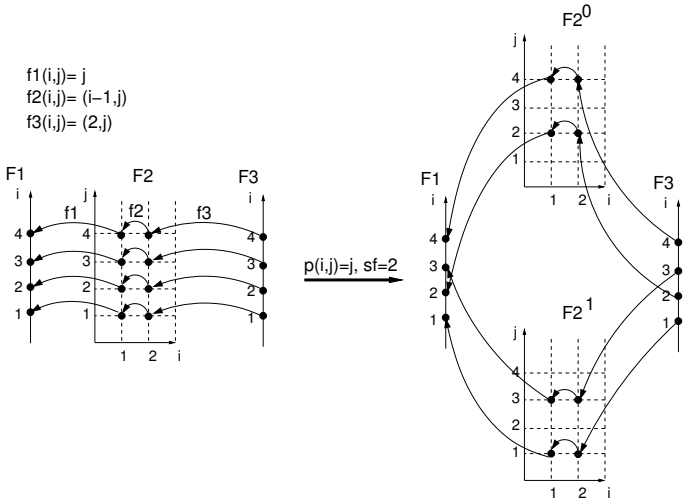


Figure 7.4: Splitting at the DDG level

for-loop of process $F2$ will be executed in process $F2^0$, whilst the odd iterations will be executed in process $F2^1$ (see Figure 7.4). We now go through the three steps in the procedure.

Step 1: As shown in the left part of Figure 7.4, process $F2$ has 4 ports that are involved in splitting: two input ports $IPD1 = \{(i,j) \in Z^2 \mid i = 1,\ 1 \leq j \leq 4\}$ and $IPD2 = \{(i,j) \in Z^2 \mid i = 2,\ 1 \leq j \leq 4\}$, and two output ports $OPD1 = \{(i,j) \in Z^2 \mid i = 1,\ 1 \leq j \leq 4\}$ and $OPD2 = \{(i,j) \in Z^2 \mid i = 2,\ 1 \leq j \leq 4\}$. As a result of the first splitting step, process $F2$ is broken into two processes $F2^0$ and $F2^1$. Additionally, based on the splitting function $p(i,j) = j$, we modify each of the embedded ports by inserting modulo if-statements: For $F2^0$ we restrict each of the ports by inserting the condition $p(i,j)\%2 = 0$, and for $F2^1$ we insert the condition $p(i,j)\%2 = 1$.

Step 2: To preserve the network functionality, additional if-statements are inserted internally to the processes $F1$, $F2$ and $F3$. This control is guarding the partitioning of the ports $IPD2, OPD1, OPD3$ and $IPD4$ which are connected to the processes $F2^0$ and $F2^1$. The conditions are obtained based on the direct and inverse dependency functions associated with each channel connected to process $F2$. This partitioning is depicted in Figure 7.5. Because all the channels are without multiplicity, the direct and inverse dependency functions are uniquely connecting a producer IP to a consumer IP. For example, for channel $Ch1$ the lexicographic minimal preimage is $lmp(i) = (1,i)$. This leads to the partitioning of $OPD1$ into two disjoint ports: $OPD11 = \{j \in Z \mid 1 \leq j \leq 4,\ j\%2 = 1\}$ and $OPD12 = \{j \in Z \mid 1 \leq j \leq 4,\ j\%2 = 1\}$, where $OPD11$ consists of iterations at which data has to be sent to process $F2^1$, and $OPD12$ consists of iterations at which data has to be sent to process $F2^0$. By applying the same procedure to all the other ports involved in splitting we arrive at the network shown in Figure 7.5.

Step 3: The network in Figure 7.5 can be further optimized. Observe how the interprocess communication between process $F2^0$ and $F2^1$ is guarded by the contradictory conditions $j\%2 = 1$ and $j\%2 = 0$. Thefore, channels $FIFO11$ and $FIFO11$ and their connecting input/output ports can be removed as they never communicate data. As a result we arrive at a network shown in Figure 7.6. Remark also that the execution of function $F2$ in process $F2^0$ and process $F2^1$ has been restricted to ips for which $j\%2 = 0$, and respectively $j\%2 = 1$.

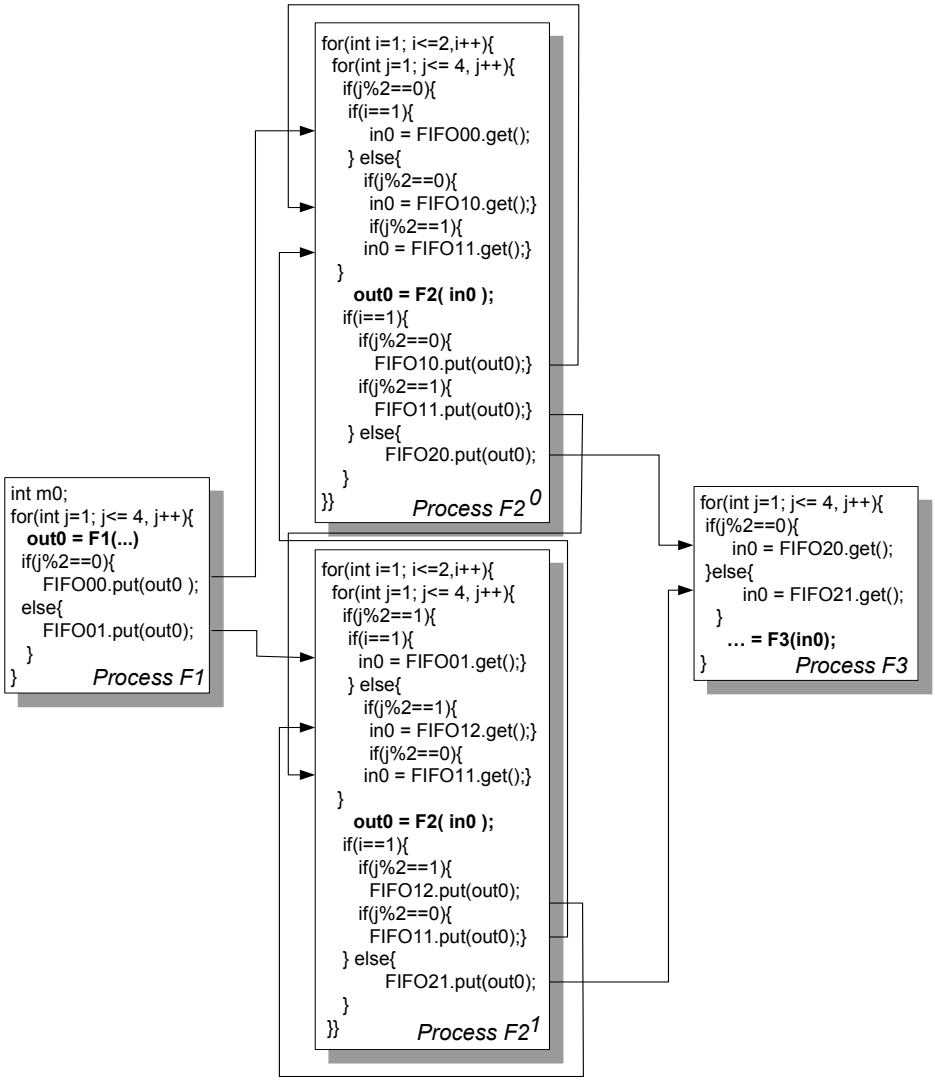


Figure 7.5: The network after process P2 was split

7.1.3 Example - splitting with multiplicity

Consider the IOM+ P/C pair of processes given in Figure 7.7. The IPD/OPD ports involved in splitting conicide with the Consumer process iteration space $IPD = \{(i,j) \in Z^2 \mid 1 \leq i \leq 4,\ 1 \leq j \leq 4\}$ and the Producer process iteration space: $OPD = \{i \in Z \mid 1 \leq i \leq 4\}$, repsectively. Suppose we choose to split the Consumer process two times (i.e.,$sf = 2$) with a partitioning function $p(j,i) = i$. As a result we get two consumer processes, $Consumer^0$ and $Consumer^1$, with two input ports affected by the splitting, $IPD1 = \{(i,j) \in Z^2 \mid 1 \leq i \leq 4,\ 1 \leq j \leq 4,\ i\%2 = 0\}$ and $IPD2 = \{(i,j) \in Z^2 \mid 1 \leq i \leq 4,\ 1 \leq j \leq 4,\ i\%2 = 1\}$, respectively. The two input ports are visualized in Figure 7.8.

For the considered P/C pair, the lexicographic maximal preimage is $lmp(i) = (1,i)$. Applying the preimage to the OPD, it turns out that for all iteration points $p(lmp(i))\%2 = 0$, so that all data would be sent to process $Consumer^1$. However, Figure 7.8 clearly shows that process $Consumer^0$ should

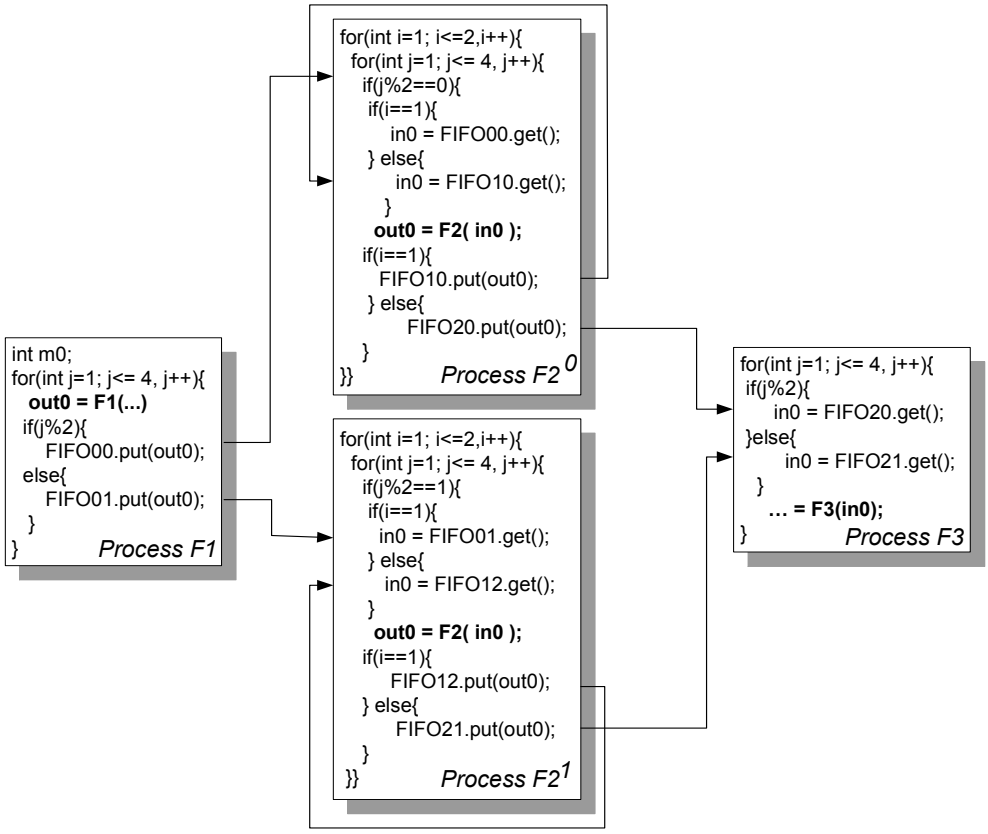


Figure 7.6: PN after removing the redundant communication

Producer:

```
for (int i=1; i<=4, i++){
    t1 = F1(...);
    FIFO.put(t1);
}
```

Consumer:

```
static t2;
for (int j=1; j<=4, j++){
  for (int i=1; i<=4, i++){
   if(i==1){
    t2 = FIFO.get();
   }
   F2(t2);
}}
```

Figure 7.7: An in-order with multiplicity P/C pair

receive data as well. Hence, because of multiplicity, the Production Restructuring has to be performed individually for IPD1 and IPD2. This gives two identical output ports $OPD1 = \{i \in Z \mid 1 \leq i \leq 4\}$, and $OPD2 = \{i \in Z \mid 1 \leq i \leq 4\}$. Next, from the lifetime analysis on the pairs OPD1/IPD1 and OPD2/IPD2, we obtain the two lmp functions $lmp1(i) = (i, 1)$ and $lmp2(i) = (i, 2)$ with their corresponding lexicographic minimal domains $IPD' = \{(i, j) \in Z^2 \mid i = 1,\ 1 \leq j \leq 4\}$, and $IPD2 = \{(i, j) \in Z^2 \mid i = 2,\ 1 \leq j \leq 4\}$, respectively. As result of the splitting we synthesize the following network structure depicted in Figure 7.9.

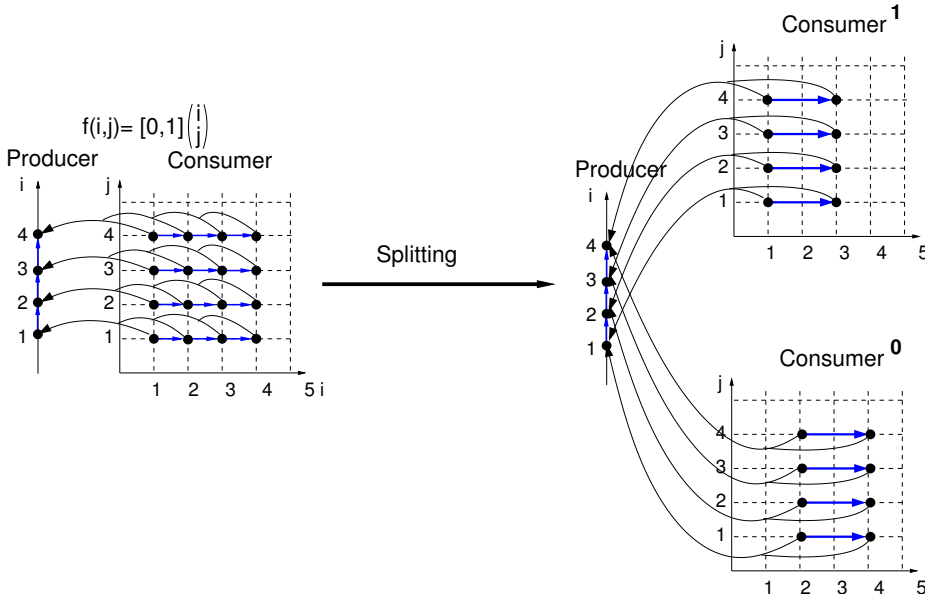


Figure 7.8: Splitting when multiplicity takes place

Producer:

```
for (int i=1; i<=4, i++){
    t1 = F1(...);
    FIFO1.put(t1);
    FIFO0.put(t1);
}
```

Consumer 1:

```
static int t2;
for (int j=1; j<=4, j++){
  for (int i=1; i<=4, i++){
    if(i%2==1){
      if(i==2){
        t2 = FIFO1.get();
      }
      F2(t2);
  }}}
```

Consumer 0:

```
static int t2;
for (int j=1; j<=4, j++){
  for (int i=1; i<=4, i++){
    if(i%2==0){
      if(i==1){
        t2 = FIFO0.get();
      }
     F2(t2);
}}}
```

Figure 7.9: The network obtained after splitting

7.2 Channel Merging

This section presents a procedure to reduce the number of in-order communication channels (IOC) in a PN. This work has been published in [20]. The reduction procedure is realized by merging IOCs belonging to the same pair of processes (adjacent FIFOs).

Channel merging is relevant when the PN is to be mapped on e.g., a multiprocessor platform that consist of processing units, a shared bus and a shared memory. In such an architecture communication and synchronization may be costly enough that redundant channels are not acceptable. Therefore, the in-order channel merging has two goals: **(1)** reducing the total memory size allocated to the network FIFO channels (Figure 7.10) and **(2)** reducing the control involved when communicating data over the network FIFO channels (Figure 7.11).

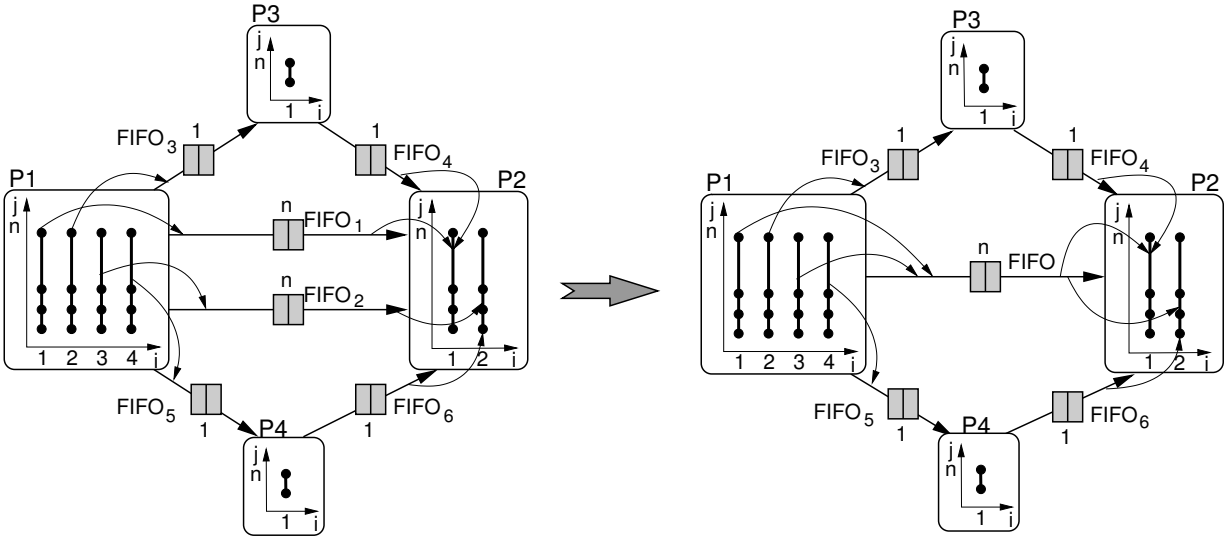


Figure 7.10: An example that shows opportunity for channel merging. We observe that $FIFO_1$ needs to be filled with n tokens before process $P2$ can start to execute its first stage in which all tokens from $FIFO_1$ are consumed. Similarly $FIFO_2$ needs to be filled with n tokens before process $P2$ can start to execute its second stage. Analyzing the flow of data through $FIFO_1$ and $FIFO_2$ (shown by arrows), we conclude that they can be merged. As a result of merging the two channels, a memory reduction of n locations is obtained.

7.2.1 Basic Terminology

Because a process in a PN is an imperative affine nested loop program, the loop nests impose an iteration domain and an ordering of the operations: FIFO *get*, function *execute*, and FIFO *put*. Figure 7.12 shows two processes communicating in a PN. The iteration domain and the lexicographic ordering are also shown. An *input/output port domain* IPD/OPD is a subset of the process iteration domain that is involved in a *put/get* interprocess communication action. Each FIFO uniquely relates an input port to an output port forming a *Producer/Consumer* pair. The Consumer is related to the Producer by a data-dependency function f that maps a consumer iteration from the IPD to the corresponding producer iteration from the OPD. As shown in Figure 7.12, a process port contains a *FIFO.get()* statement or a *FIFO.put()* statement. In order to reason about the FIFO merging we need to be able to keep track of the order of executing such *get/put* statements within a single process.

Observe that for operations belonging to the same port (*IPD/OPD*) the lexicographic order represents a total order (i.e., every two operations can be ordered by means of lexicographic order). However, the channel merging involves ordering of statements belonging to different ports. This ordering can not be obtained relying solely on the lexicographic order. To impose a *total order* among statements from different ports we make use of the total order imposed between arbitrary program statements as described in Chapter 2.

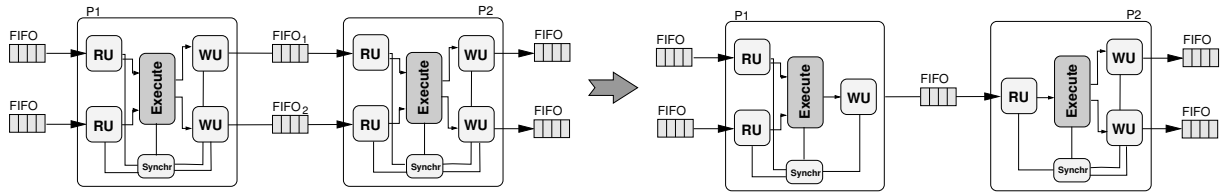


Figure 7.11: An example that shows opportunity for control reduction due to channel merging. For each FIFO we have a read control unit RU and a write control unit WU. The structure of these control units depends greatly on the implementation. For example, in a hardware implementation [113] they may contain counters used to address the FIFO locations, and automatons used to check the status of the FIFO (empty or full). By merging the two FIFOs we reduce also the control units.

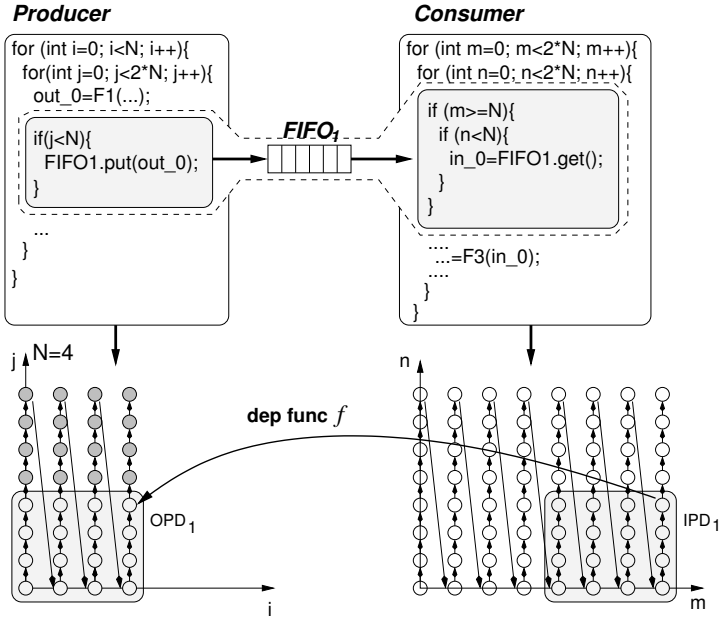


Figure 7.12: A P/C pair. For the Producer one output port domain (OPD_1) and for the Consumer the input port domain (IPD_1) are shown. OPD_1 is connected to IPD_1 via $FIFO_1$.

7.2.2 Problem Statement

The PNs generated by Compaan often contain multiple FIFO channels between two nodes, i.e., adjacent FIFOs (see Figure 7.14). The problem addressed here consists in reducing the number of adjacent FIFO channels by merging some of them together. To merge a group of k adjacent FIFOs: $Ch_1(OPD_1, IPD_1, f_1)$, ... $Ch_k(OPD_k, IPD_k, f_k)$, the total order of production in the domain $\cup_{i=1}^{k} OPD_i$ must be the same as the total order of consumption in the domain $\cup_{i=1}^{k} IPD_i$. If the total orders of production and consumption are not the same, it may be possible to enforce the textual order at the producer or consumer side in such a way that in the end the total orders will be the same. In this case, we can merge the group of k FIFOs by replacing them with a single one. To have an understanding of the problem, we present an illustrative example containing various situations related to the in-order channel merging.

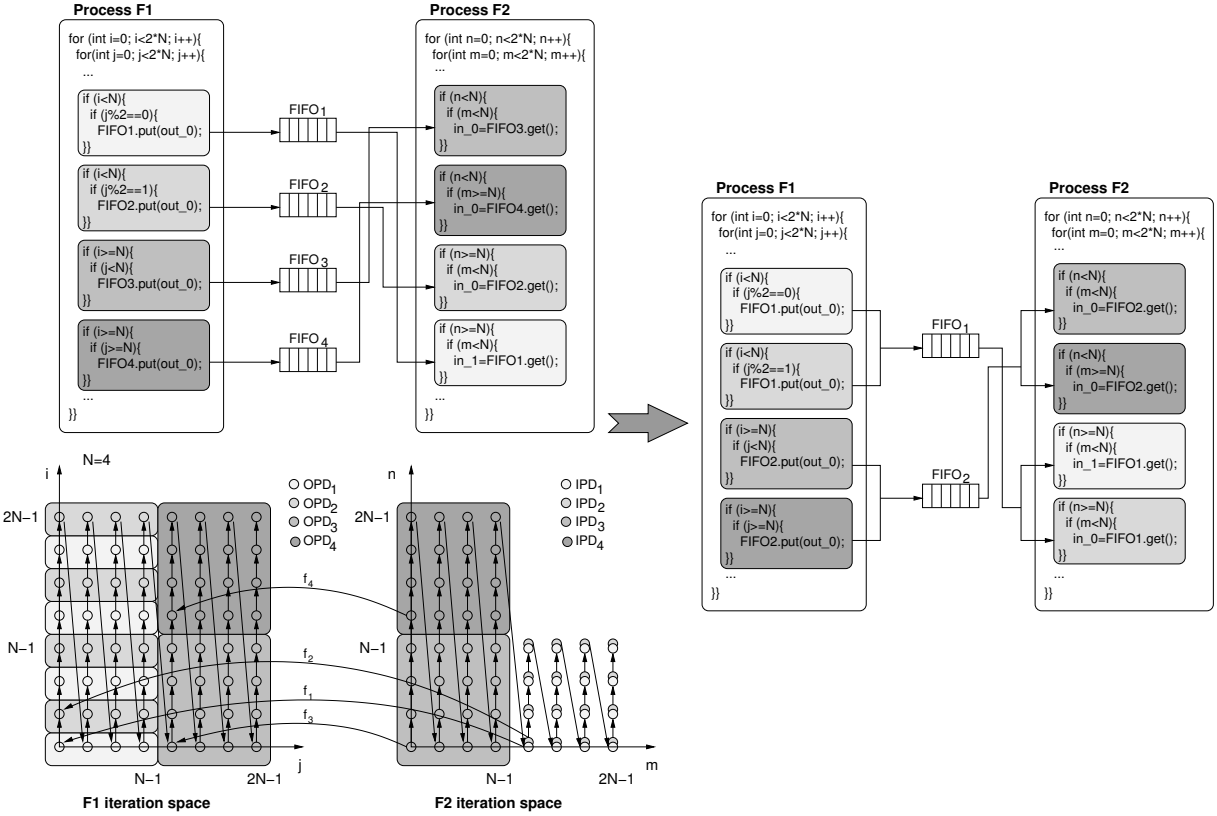


Figure 7.13: A comprehensive example showing various situations related to the in-order channel merging problem.

Figure 7.13 shows two processes that are communicating data over four adjacent FIFO channels. First we analyze $FIFO_3$ and $FIFO_4$. We observe that the total order of producing tokens in OPD_3 and OPD_4 is the same as the total order of consuming tokens in IPD_3 and IPD_4. This leads to the conclusion that the two FIFO channels can be replaced by a single one. Further we analyze $FIFO_2$ and $FIFO_3$. Here the total order of producing tokens in OPD_2 and OPD_3 is not the same as the total order of consuming tokens in IPD_2 and IPD_3: Tokens are first produced in OPD_2 and then in OPD_3 but are first consumed in IPD_3 and then in IPD_2. As a consequence, the two FIFO channels cannot be replaced by a single FIFO channel. The same situation takes place for the pairs $FIFO_1$ and $FIFO_3$, $FIFO_1$ and $FIFO_4$, and $FIFO_2$ and $FIFO_4$. Consider now $FIFO_1$ and $FIFO_2$. Here the total order of producing tokens in OPD_1 and OPD_2 is not the same as the total order of consuming tokens in IPD_1 and IPD_2. Tokens are produced first in OPD_1 and then in OPD_2 but are first consumed in IPD_2 and then in IPD_1. At a first glance we would say that FIFO merging is not possible. However, we observe that at the consumer side the order of consuming tokens from the channels is given only by the textual order of IPDs. Hence, we can control the order of consuming by controlling the textual order. This leads to interesting results: Even if initially the merging was not possible, after a simple code transformation (textual reordering) - we impose the order from the producer side to the consumer side - the FIFOs can be merged. This situation is illustrated at the right side of Figure 7.13. The textual reordering is done as follows: At the consumer side we change the textual order of reading the channels putting first OPD_1, and then OPD_2. As a result we can merge the two FIFO channels.

In the next section we describe in detail the solution to the merging problem. First we introduce the *Basic FIFO Merging Test (BFMT)*, which is used to decide whether or not two adjacent FIFOs can be merged. Next, we extend this test to a *General FIFO Merging Test (GFMT)*, which is used to find out whether or not more than two adjacent FIFOs can be merged. Finally, we provide the procedure that uses these tests to find all the groups of adjacent FIFOs that can be merged within a PN.

7.2.3 Empty Domain Test

To present our solution we make use of the Empty Domain Test (ET) introduced in Section 2.3.4. Recall that ET is a boolean test used to decide whether a given domain D contains integer points:

$$ET(D) = \begin{cases} True, & \nexists x \in D \\ False, & \exists x \in D \end{cases}$$

In our presentation a domain D is a union of convex domains, typically given as: $D = \{(x, y) \mid x \in IPD_1, y \in IPD_2, x \prec_t y\}$, where "$\prec_t$" denotes the total order (see Chapter 2). Next we introduce two basic properties of this test that are useful for our future presentation:

Proposition 11 *If* $D_1 \subset D_2$ *then* $ET(D_1) = ET(D_1) \vee ET(D_2)$.

Proposition 12 $ET(D_1 \cup D_2) = ET(D_1) \wedge ET(D_2)$.

7.2.4 Basic FIFO Merging Test

We define the Basic FIFO Merge Test as a function that accepts as input arguments two adjacent FIFO channels Ch_1 and Ch_2, and returns a boolean: *True* when these FIFOs can be merged and *false* when they cannot be merged.

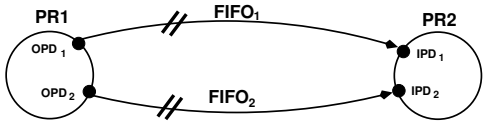


Figure 7.14: PN showing opportunity in FIFO channels sharing

Figure 7.14 shows a P/C pair of processes connected by two FIFO channels $Ch_1 = (OPD_1, IPD_1, f_1)$ and $Ch_2 = (OPD_2, IPD_2, f_2)$. In order to merge Ch_1 and Ch_2, the total order of producing and consuming over the two channels must be the same. This means that the total order of producing data at iterations included in $OPD_1 \cup OPD_2$ must be the same as the total order of consuming data at iterations included in $IPD_1 \cup IPD_2$. Formally this statement can be verified by checking whether the domains specified by the following two sets of constraints are empty:

$$\mathbf{PD}: \begin{cases} x \in IPD_1, & (1) \\ y \in IPD_2, & (2) \\ x \prec_t y, & (3') \\ f_1(x) \succ_t f_2(y), & (4') \end{cases} \qquad \mathbf{DD}: \begin{cases} x \in IPD_1, & (1) \\ y \in IPD_2, & (2) \\ x \succ_t y, & (3'') \\ f_1(x) \prec_t f_2(y), & (4'') \end{cases}$$

where f_1 and f_2 are the data-dependency functions associated with the two channels. Note that, as introduced in Chapter 2, "$\prec_t$" denotes the total order, "$\prec$" denotes the lexicographical order, while the textual order is defined using a boolean function T. If there are iterations x and y that obey one of the constraints defined above, then merging is not possible. Therefore, the BFMT can be expressed as follows:

$$BFMT(Ch_1, Ch_2) = ET(PD) \wedge ET(DD).$$

Next, we investigate how BFMT is solved in the PN context, i.e., when operating over convex input and output port domains connected via affine dependency functions. To solve ET we use ILP [37, 73]. To arrive to an ILP formulation we first decompose the relations $(3'), (4')$ and $(3''), (4'')$ according to relation (2.1) given in Chapter 2.

$$PD : \begin{cases} x \in IPD_1, & (1) \\ y \in IPD_2, & (2) \\ x \prec_t y, & (3') \\ f_1(x) \succ_t f_2(y). & (4') \end{cases} =$$

$$PD_1 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x \prec y, \\ f_1(x) \succ f_2(y). \end{cases} \cup \quad PD_2 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x \prec y, \\ f_1(x) = f_2(y) \wedge T(f_2(x), f_1(y)) = 1. \end{cases} \cup$$

$$PD_3 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x = y \wedge T(x, y) = 1, \\ f_1(x) \succ f_2(y). \end{cases} \cup \quad PD_4 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x = y \wedge T(x, y) = 1, \\ f_1(x) = f_2(y) \wedge T(f_2(x), f_1(y)) = 1. \end{cases}$$

And similarly for the DD domain follows:

$$DD : \begin{cases} x \in IPD_1, & (1) \\ y \in IPD_2, & (2) \\ x \succ_t y, & (3') \\ f_1(x) \prec_t f_2(y). & (4') \end{cases} =$$

$$DD_1 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x \succ y, \\ f_1(x) \prec f_2(y). \end{cases} \cup \quad DD_2 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x \succ y, \\ f_1(x) = f_2(y) \wedge T(f_1(x), f_2(y)) = 1. \end{cases} \cup$$

$$DD_3 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x = y \wedge T(y, x) = 1, \\ f_1(x) \prec f_2(y). \end{cases} \cup \quad DD_4 : \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x = y \wedge T(y, x) = 1, \\ f_1(x) = f_2(y) \wedge T(f_1(x), f_2(y)) = 1. \end{cases}$$

Based on Proposition 12 and using the previous two decompositions, we can evaluate the BFMT as follows:

$$\begin{aligned} BFMT = \quad & ET(PD_1) \wedge ET(PD_2) \wedge ET(PD_3) \wedge ET(PD_4) \wedge \\ & ET(DD_1) \wedge ET(DD_2) \wedge ET(DD_3) \wedge ET(DD_4). \end{aligned}$$

The description of the domains PD_2, PD_3, PD_4, DD_2, DD_3 and DD_4 depends on the textual order. Hence, we further decompose each of those sets into two subsets: The first set D_i^{lex} is defined purely according to the lexicographic order and therefore we can apply ILP in order to evaluate $ET(D_i^{lex})$. The second set D_i^{Text} is defined only according to the textual order. Thus, $D_i = D_i^{lex} \cap D_i^{Text}$, where $i = \overline{2,4}$. For example according to this decomposition the domain $PD2$ is decomposed as follows: $DD_2 = DD_2^{lex} \cap DD_2^{Text}$, where

$$DD_2^{lex} = \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ x \succ y, \\ f_1(x) = f_2(y), \end{cases} \quad \text{and} \quad DD_2^{Text} = \begin{cases} x \in IPD_1, \\ y \in IPD_2, \\ T(f_1(x), f_2(y)) = 1. \end{cases}$$

Now $T(f_1(x), f_2(y)) = 1 \Rightarrow DD_2^{lex} \subset DD_2^{Text}$ such that, applying Proposition 11, $ET(DD_2) = ET(DD_2^{lex}) = ET(DD_2^{lex}) \vee ET(DD_2^{Text})$(1). On the other hand $T(f_1(x), f_2(y)) = 0 \Rightarrow DD_2^{Text} = \emptyset \Rightarrow DD_2^{Text} \subset DD_2^{lex}$, whence, $ET(DD_2) = ET(DD_2^{Text}) = ET(DD_2^{lex}) \vee ET(DD_2^{Text})$(2). (1) and (2) imply that $\forall\, T(f_1(x), f_2(y))$:

$$ET(DD_2) = ET(DD_2^{lex}) \vee ET(DD_2^{Text}).$$

Similar relations can be established for PD_2, PD_3, PD_4, DD_3 and DD_4. As a result we get the following BFMT formula:

$$\begin{aligned} BFMT = \quad & ET(PD_1) \wedge \\ & (\ ET(PD_2^{lex}) \vee ET(PD_2^{Text})\ ) \ \wedge \\ & (\ ET(PD_3^{lex}) \vee ET(PD_3^{Text})\ ) \ \wedge \\ & (\ ET(PD_4^{lex}) \vee ET(PD_4^{Text})\ ) \ \wedge \\ & ET(DD1) \wedge \\ & (\ ET(DD_2^{lex}) \vee ET(DD_2^{Text})\ ) \ \wedge \\ & (\ ET(DD_3^{lex}) \vee ET(DD_3^{Text})\ ) \ \wedge \\ & (\ ET(DD_4^{lex}) \vee ET(DD_4^{Text})\ ). \end{aligned}$$

We thus have decomposed the Basic FIFO Merge Test into separate tests for the lexicographic order and the textual order. The FIFO merging is possible (i.e., $BFMT = true$) iff all the component predicates are $true$:

$$\begin{aligned} BFMT = true \quad \equiv \quad & ET(PD1) = true \wedge \\ & (\ ET(PD_2^{lex}) \vee ET(PD_2^{Text})\ ) = true \ \wedge \\ & (\ ET(PD_3^{lex}) \vee ET(PD_3^{Text})\ ) = true \ \wedge \\ & (\ ET(PD_4^{lex}) \vee ET(PD_4^{Text})\ ) = true \ \wedge \\ & ET(DD1) = true \wedge \\ & (\ ET(DD_2^{lex}) \vee ET(DD_2^{Text})\ ) = true \ \wedge \\ & (\ ET(DD_3^{lex}) \vee ET(DD_3^{Text})\ ) = true \ \wedge \\ & (\ ET(DD_4^{lex}) \vee ET(DD_4^{Text})\ ) = true. \end{aligned}$$

Here we make the observation that while the lexicographic order is fixed and, say, $ET(PD_i^{lex}) = false$, the textual order can be easily modified in such way that $ET(PD_3^{Text}) = true$. Hence, $ET(PD_i^{lex}) \vee ET(PD_3^{Text}) = true$. *In other words, although the total order does not allow to merge the channels, by choosing an appropriate textual order the merging becomes possible*. Therefore, we first evaluate the predicates involving lexicographic order and then choose textual orders for the IPDs and OPDs trying to make the merging possible. At the end of the evaluation procedure we get a list of constraints regarding the textual order of OPDs and IPDs. We analyze whether the list of textual constraints is logically valid and if yes, we change the internals of the processes accordingly. The following example presents a case in which the list of textual constraints are contradictory:

Example: Consider the hypothetical case where we have evaluated the Empty Domain Test on the domains with respect to the lexicographic order, getting $ET(PD_3^{lex}) = false$ and $ET(DD_3^{lex}) = false$. To make the merging possible, we impose the following textual orders: From the first relation, IPD_2 must textually precede IPD_1, and from the second relation, IPD_1 must textually precede IPD_2: $ET(PD_3^{Text}) = true$ and $ET(DD_3^{Text}) = true$. But this set of textual constraints is contradictory. Hence, we conclude that the merging is not possible. □

7.2.5 Extending the Merge Test to more than two FIFO channels

So far we have analyzed only the case in which two FIFO channels are considered for merging. For this purpose we have introduced the BFMT test. However, we still have to solve the merging problem in the general case where there are more than two adjacent FIFOs. For this purpose we introduce the General FIFO Merge Test (GFMT).

Proposition 13 : *Consider three FIFO channels Ch_1,Ch_2 and Ch_3 with their dependency functions f_1, f_2 and f_3, respectively, that are connecting the same P/C pair of processes. Hence, Ch_1,Ch_2 and Ch_3 can be merged ($GFMT(Ch_1, Ch_2, Ch_3) = true$) iff:*

$$BFMT(Ch_1, Ch_2) = true \ \wedge \ BFMT(Ch_1, Ch_3) = true \ \wedge \ BFMT(Ch_2, Ch_3) = true.$$

Proof: '$\Rightarrow$': Consider three arbitrary points $x \in IPD_1$, $y \in IPD_2$ and $z \in IPD3$. Because "$\prec_t$" is a total order it turns out that we can order x, y and z. Suppose that $x \prec_t y \prec_t z$. On the other hand $BFMT(Ch_1, Ch_2) = true$, such that $x \prec_t y \Rightarrow f_1(x) \prec_t f_2(y)$. Applying a similar procedure for the other two pairs it turns out that: $f_1(x) \prec_t f_2(y) \prec_t f_3(z)$. Hence, the channels can be merged: $GFMT(Ch_1, Ch_2, Ch_3) = true$.

'$\Leftarrow$': "$\prec_t$" is a total order that does not get disturbed from the Consumer to the Producer process (i.e., the three FIFOs can be merged). This implies that also restricted to pairs of IPDs and OPDs the order does not get disturbed such that each pair of channels can be separately merged. □

Based on the following corollary we can decide whether n FIFOs can be merged into a single one.

Corollary 3 *Consider n FIFO channels Ch_i with $i = \overline{1,n}$ connecting the same P/C pair of processes. Hence,*

$$GFMT(Ch_1, .., Ch_n) = true \Leftrightarrow \ \forall\, u, v, \ 1 \leq u, v \ \leq n, BFMT(Ch_u, Ch_v) = true.$$

Remark 1 $C_n^2 = \frac{n!}{(n-2)!2!} = \frac{(n-1)n}{2}$ BFMT tests have to be evaluated to prove that $GFMT(Ch_1, ..., Ch_n) = true$.

To decide whether n adjacent FIFOs can be merged, we have to calculate C_n^2 BFMTs. In case not all the tests return true, we divide the original group of n FIFOs in groups of FIFOs for which $GFMT$ is true. To arrive at a minimum number of FIFOs, we have to find the largest, exclusive groups of FIFOs, such that: For each group $GFMT$ is true, all channels belong at least to one group, no channel belong to two groups, and each group is maximal. To solve this problem we make use of a graph formulation. Thus, we build a graph where each FIFO represents a vertex and each $BFMT(Ch_i, Ch_j)$ that is $true$ represents an edge between vertexes Ch_i and Ch_j. Therefore, our problem is reduced to finding all the maximal and exclusive complete subgraphs (cliques) in the graph. For solving this problem we can use an expensive algorithm that explores all the possibilities of complete subgraphs [88] or we can use a heuristic algorithm [14, 32].

7.2.6 Implementation and Results

In Compaan, we have implemented a transformation which changes the topology of a PN by merging adjacent FIFOs, thus reducing the number of in-order channels in the network. The transformation makes extensive use of polyhedron manipulations, and integer linear programming supported by existing C libraries, such as PolyLib [105], Pip [37] or Omega [73].

In Table 7.1, we present some quantitative results when applying FIFO merging to seven PNs generated by Compaan. For each application, we give the number of processes, the total number of channels before merging, the number of FIFOs before merging, the number of FIFOs after merging, and the reduction rate. We observe from this table that the reduction rate depends greatly on the original process network. The PN that exhibit the highest reduction rate (SVD and M-JPEG) initially contain a larger number of adjacent FIFOs while the PNs that have a reduction rate of 0 % have none or very few adjacent FIFOs. But this is not always true: The Motion Estimation PN has a large number of adjacent FIFOs, but a very low reduction rate (only 5 %). We draw the conclusion that the reduction rate not only depends on the topology of the PN, but also on the algorithm implemented by this PN.

Algorithm name	Nb. Processes	Initial Nb. Channels	Initial Nb. FIFOs	Nb. of FIFOs after merging	FIFO reduction
SVD	8	118	84	35	58 %
Faddeev	9	28	24	19	20 %
M-JPEG	9	50	33	13	60 %
Motion Estim	11	98	98	93	5 %
DigBeamFormer	8	17	17	14	17 %
QR	5	12	12	12	0 %
Gauss-Elimin	4	11	7	7	0 %

Table 7.1: Experimental results

Chapter 8

Conclusions and Future Research

The thesis presents a compilation approach that enables the conversion of the complete class of static affine nested loop programs to a multi-threaded representation specified as process network. Our approach relieves the user from having to deal with the low-level details of program iteration space partitioning, data sharing, and thread synchronizations. The approach is analytical; there is not a single heuristic involved. We have shown how the compilation approach can be divided into four main steps. We have presented the main idea behind each step in a separate chapter, how to formulate the step as an Integer Linear Programming (ILP) problem and how to solve it using ILP or alternative less computational complex techniques. Furthermore, we have illustrated each step using a running example.

The steps and techniques presented have been implemented in the Compaan compiler, replacing, extending and generalizing the steps presented in [58]. The only parts that are missing a software implementation are the Life Time Analysis (partly implemented) and the Process Splitting. The examples given in this thesis are generated by this compiler. We also showed the results we get from running Compaan on a set of 8 applications from the area of signal and image processing. As shown in [84] and further enhanced in [68, 69, 87], the Compaan compiler allows to further develop a programming environment for heterogeneous multiprocessor platforms. Using PNGen, one can obtain software implementations for PNs that can be mapped and executed on CPUs or DSPs. On the other hand, using Laura [109], one can also obtain hardware implementations for PNs making use of dedicated IP cores and reconfigurable hardware. An arbitrary mix between hardware and software is also possible [69].

8.1 Compilation Flow Overview and Results

The steps presented in the beginning of the thesis in Figure 1.3 are implemented in a compiler tool chain shown in Figure 8.1. The first tool, called MatParser [57], performs an exact data-dependence analysis which is used for the Consumption Restructuring step. Additionally, the tool is able to convert the input program to a single assignment format. The *Process Network Generator* tool, or *PNGen*, implements the remaining four steps presented in this thesis and generates a PN description. PNGen replaces the *Panda* tool in Compaan as presented in [58]. The user can choose the PN to be generated in C++ or in Java. The generated code allows us to simulate the PN and to verify that the PN is equivalent to the original sequential program. It is also possible to generate hardware for a PN. The *Laura* tool [113] transforms the network generated by *PNGen* into an equivalent VHDL description that can be synthesized and mapped on an FPGA platform. The five compilation steps in Figure 8.1 make

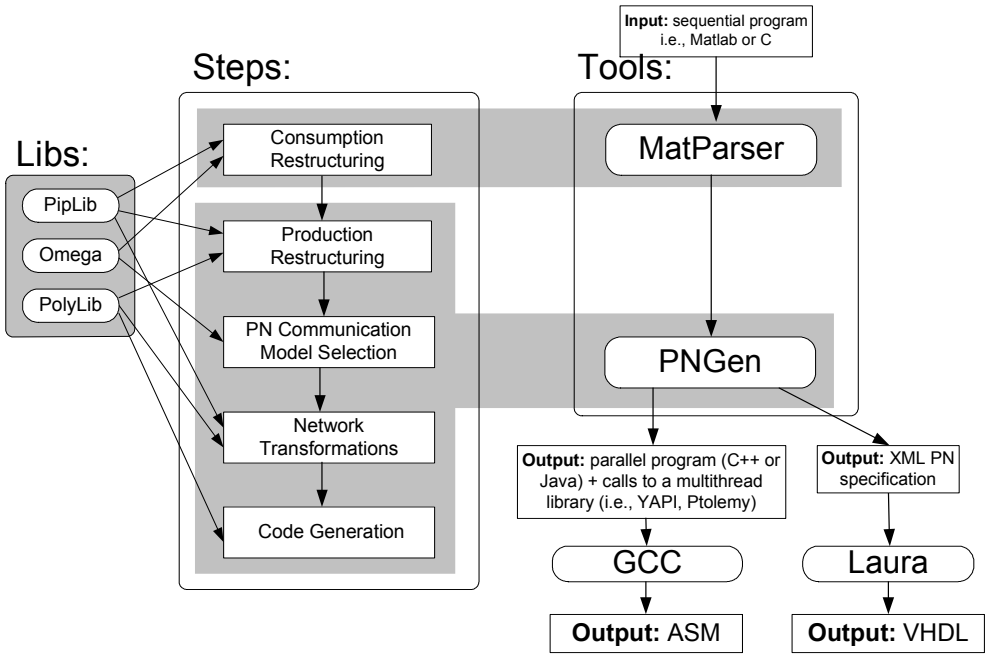


Figure 8.1: The Compaan compiler flow overview

extensive use of polyhedron manipulations, matrix decompositions, and integer linear programming. To perform these operations we rely on open source software libraries, like PolyLib [62, 105], Pip [37], and Omega [73].

8.1.1 Experimental Results

In Table 8.1, we present some quantitative characteristics obtained form compiling 8 applications, of which the M-JPEG case is described separately in [84] and the QR case in [48]. For each application we have given the number of lines of the original sequential specification, the compilation time required on a Pentium III processor, and the number of processes and channels generated. Table 8.1 also shows the P/C communication types. Based on this data, we observe that in approximately 90% of the P/C pairs, a communication structure based on a FIFO buffer is sufficient.

Algorithm name	Nb. Lines of code	Compilation time (M:S)	Nb. Processes	Nb. Channels	Channels type IOM-/IOM+/OOM-/OOM+
LU-Factor	30	00:40	5	26	14 / 6 / 5 / 1
QR-Decomp	26	00:13	6	12	12 / 0 / 0 / 0
SVD	65	05:34	8	69	35 / 4 / 30 / 0
Faddeev	35	00:27	9	23	19 / 3 / 1 / 0
Gauss-Elimin	26	00:14	4	11	7 / 0 / 1 / 3
BeamFormer	12	01:09	8	14	14 / 0 / 0 / 0
Motion Estim	78	01:57	11	93	93 / 0 / 0 / 0
M-JPEG	43	01:04	9	30	13 / 17 / 0 / 0

Table 8.1: Experimental results

In the remaining 10% of the cases, we need to realize a reordering at the Consumer process, using a RAM or CAM memory structure. The output of the Compaan tool chain has been further quantified by running the Laura VHDL backend. [109] presents a complex case study that quantifies the performance of the VHDL representation of the PN obtained after compiling a Beam-Forming application. This application consists of three algorithms (i.e., Matrix-Matrix multiplication, QR factorization, and SVD decomposition) which are representative for the class of applications processed by Compaan.

8.2 Directions for Further Research

Although the Compaan compiler has arrived at a stage where case studies can be performed, there is room to further improve its output PNs. We have identified two future research directions, each of them begin further discussed separately. They are source code transformations and memory estimations.

8.2.1 Source code transformations

IF-conversion to enlarge the class of Compaan input applications: The class of applications that Compaan accepts for compilation is quite restrictive, and does not include applications with data dependent control. The reason behind the restriction is the incapability of the dependence analysis (used in the context of this work) to be applied in the presence of any other constraints different than linear inequalities control flow dependencies. A control dependence exists between two statements when the execution of one statement can prevent the execution of the other. Nevertheless, there is a classic technique developed in the 80s called *if-conversion* that converts all of the control dependencies into data dependencies [2, 56]. The conversion is done by introducing logical variables to control the execution of statements in the program. The technique has been developed to convert sequential programs to a form more suitable for vector machines in order to allow more vector statements to be generated out of sequential code. For example the control dependency `if(r[i]>0)` present in the next piece of code is not accepted by Compaan:

```
1.for (int i=1; i<=N; i++){
2.    r[i]=Init(...);
3.}
4.for (int i=1; i<=N; i++){
5.   if(r[i]>0)
6.     r[i]=F(r[i]);
7.}
8.for (int i=1; i<=N; i++){
9.    Sink(r[i]);
10.}
```

By applying if-conversion on the assignment statement present in line 6., we get a guarded assignment statement containing an extra input parameter corresponding to the original control. Thus, as shown below the program becomes analyzable by our compiler:

```
for (int i=1; i<=N; i++){
    r[i]=Init(...);
}
for (int i=1; i<=N; i++){
    r[i]=FP(r[i]);
}
for (int i=1; i<=N; i++){
    Sink(r[i]);
}
```

```
FP(r[i]){
   if(r[i]>0)
     return F(r[i]);
   else
     return r[i];
}
```

,where the function F has been replaced by the function FP:

The network generated by Compaan when running the new application is made of three processes, among which the central one actions as a copy node in case `r[i]>0`, but performs the function F otherwise. To support this feature, special support has to be provided at the hardware code-generation step.

Vectorization to reduce the inter process communication: The performance of executing a Compaan network very much depends on the characteristics of the target multiprocess architecture. In certain cases where the target architecture provides physical support for the network communication channels, the communication may not be a problem. However, in case of a shared memory multiprocessor architecture, the excess of inter process communication may kill the execution. Thus, in order to still make use of the available PN parallelism, a reduction of the communication seams to be a potential solution. As shown by the next example, the communication can be reduced by vectorizing either the input source code [70, 7] or the processes loop bodies. Currently, from the next piece of code Compaan generates a PN consisting of two processes that communicate $4N$ tokens.

```
for (int i=0; i<=4*N; i++){
    r[i]=Init(...);
}
for (int i=0; i<=4*N; i++){
    Sink(r[i]);
}
```

Nevertheless, after vectorizing each of the two loops by factor 4 the following codes results:

```
for (int i=0; i<=N; i++){
    R[i]=Init(...);
}
for (int i=0; i<=N; i++){
    Sink(R[i]);
}
```

As a result of the vectorization, an element in the array `R` represents a packet of the original array `r`, i.e., `R[i]=(r[4*i],r[4*i+1],r[4*i+2],r[4*i+3])`. Thus, the generated PN will communicate four times less tokens. If the number of loop iterations is not a multiple of the vector factor then in order to avoid the generation of extra processes (induced by the loops scalar epilogues), we can introduce few extra iterations for which no operation takes place.

8.2.2 Memory estimates

According to Section 6.2.1, a PN in which the communication channels use the same amount of memory as the corresponding memory arrays present in the sequential code obtained after the first two compilation steps, represents a PN free of deadlock. Nevertheless, the resulted memory (although not in a single assignment format) may be too large when targeting a hardware platform with limited storage resources. Thus, establishing a compile-time technique to minimize the PN storage requirements seams to be an interesting future research direction. Preliminary results for establishing a PN storage lower bound are given in Appendix .

Appendix: Memory estimates

In this appendix we investigate the PN storage lower bound. We establish a condition regarding the channels memory sizes for avoiding network deadlock. However, the condition is only necessary such that further work has to be done in order to establish the sufficient condition.

As presented in Section 6.2.1, a PN in which every communication channel uses the same amount of memory as the corresponding memory arrays in the sequential code obtained after the first two compilation steps, represents a PN free of deadlock. Nevertheless, the resulting memory (although not single assignment) may be too large when targeting a platform with limited storage resources.

In general, for an arbitrary KPN specification (with data-dependent selection of the active IPDs/ OPDs) the problem of determining the storage capacities of the communication channels needed to avoid deadlock is undecidable [15, 16]. However, the Compaan PN description is a particular one, without data-dependent consumption, production, or execution. Additionally the network processes are firing over polyhedral input/output port domains.

Next we introduce three important notions: *deadlock*, *data-path*, and *adjacent data-paths*, useful in our presentation:

Definition 9 *A PN is in deadlock if all the processes are blocked. A deadlock is artificial if at least one process blocks on a full FIFO that has to be enlarged in order the network to continue the execution.*

A PN may artificial deadlock because some of the channel sizes are too small. As presented in [41, 9, 71], an artificial deadlock can be resolved by enlarging the full channels using different run-time approaches. Of course by bounding the channel sizes to the limit, the execution of the network may slow down. Nevertheless, the slow down may be compensated by the fact that on the target platform (in our case an FPGA) the processes run concurrently.

Consider a PN with whose processes iterate over polyhedral input/output port domains.

Definition 10 *We define a* **data-path (dp)** *as a tuple:*

$$dp \;=\; < (d_i, P_j),\ (d_k, P_l) >,$$

where data produced by process P_j at an iteration belonging to the polyhedral domain d_i is after perhaps passing through intermediary processes consumed by the process P_l at an iteration belonging to the domain d_k.

In other words, there is a data-path between process P_j and process P_l if at least a token produced by P_j is consumed by process P_l. For example, in the network depicted in Figure 8.2, we explicitly

show three of the possible data paths. One of them is $dp_2 =< (d_1, PR_2), (d_2, PR_6) >$, where d_1 is a subset of the output port domain OPD_1 belonging to process PR_2, $d_1 \in OPD_1$, and d_2 is a subset of the input port domain IPD_2 belonging to process PR_6, $d_2 \in IPD_2$. *In this case, a* **necessary condition** *for the existence of this data path is that inside process PR_3, OPD_2 and IPD_1 are not disjoint.*

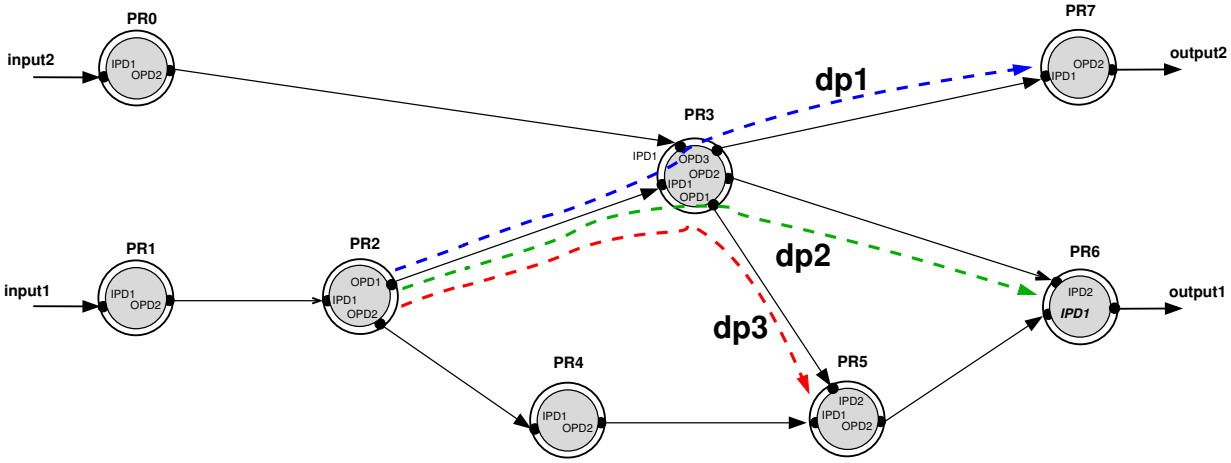


Figure 8.2: Example of a data-path in a KPN

Based on the notion of data-paths, we now introduce the notion of *adjacent data-paths*:

Definition 11 *We say that two data-paths $dp_1 =< (a_1, A_1), (a_2, A_2) >$ and $dp_2 =< (b_1, B_1), (b_2, B_2) >$ are* adjacent, *$dp_1 \ \delta \ dp_2$, if the next three conditions are simultaneously true:*

$$\begin{cases} A_1 = B_1, \\ A_2 = B_2, \\ a_2 \cap b_2 \neq \varnothing. \end{cases}$$

The first two conditions express that two adjacent data-paths relate the same P/C pair of process. Additionally, the third condition expresses the fact that the firing of a Consumer iteration ip belonging to the intersection of the domains a_2 and b_2 (i.e., $ip \in a_2 \cap b_2$) is conditioned by the firing of two different Producer iterations.

Making the PN data-paths explicit

Next we present a procedure to establish the data-paths existing within a PN, i.e., to derive the producer domain d_i and the consumer domain d_k of every data-path $dp =< (d_1, P_1), (d_k, P_k) >$.

Suppose that between process P_1 and process P_k there are $k-2$ intermediate processes through which tokens are passing. Each pair of direct connected processes $< P_t, P_{t+1} >$ constitutes a P/C pair in which the process P_t at iterations included in the output port domain opd_{P_t} produces data which is consumed by process P_{t+1} at iterations included in the input port domain $ipd_{P_{t+1}}$. The relation between the producer opd_{P_t} and the consumer $ipd_{P_{t+1}}$ is defined by an affine mapping M_t. Suppose

the consumer domain is an n-dimensional input port domain $ipd_{P_{t+1}}$. Then opd_{P_t} is represented by the mapping of the integral points included in $ipd_{P_{t+1}}$ through the mapping M_t:

$$opd_{P_t} = M_t(ipd_{P_{t+1}}). \tag{8.1}$$

For all the P/C pairs belonging to dp the next sequence of mappings connects ipd_{P_k} to opd_{P_1}:

$$opd_{P_1} \overset{M_1}{\longleftarrow} ipd_{P_2}, opd_{P_2} \overset{M_2}{\longleftarrow} \quad ... \quad \overset{M_{k-2}}{\longleftarrow} ipd_{P_{k-1}}, opd_{P_{k-1}} \overset{M_{k-1}}{\longleftarrow} ipd_{P_k}. \tag{8.2}$$

In this way the domain d_1 given by,

$$d_1 = ipd_{P_1} \cap M_1(ipd_{P_2} \cap ... \cap M_{k-1}((ipd_{P_{k-1}} \cap M_{k-1}(ipd_{P_k}))...). \tag{8.3}$$

contains all the iterations belonging to process P_1 at which there are produced tokens further consumed by process P_k. Due to the operations involved in deriving d_1 (i.e. polyhedral images and intersections) it turns out that d_1 can be represented as a union of disjoint polytopes. By making abstraction of the intermediate processes it can stated that:

Proposition 14 *Let $M = M_1 \circ M_2 ... \circ M_{k-1}$ and let $\mathcal{M}^{-1}$ be the lexicographically maximal preimage of M relatively to the producer d_1 domain, then,*

$$d_1 = M(d_k), \tag{8.4}$$
$$d_k = \mathcal{M}^{-1}(d_1). \tag{8.5}$$

The rank and read functions relative to a data-path

Definition 12 *Let P be the union of N polytopes P_i, $P = \cup_{i=1}^{N} P_i$. The rank of an integral point x relative to P, $rank(x, P)$, is the function giving the number of integer points included in P that are lexicographically preceding x (i.e., $x' \prec x$). Hence, $rank(x, P) = \sum_{l=1}^{d} |J_l(x)| + 1$, where*

$$J_l(x) = \{x^{'} \in P \cap Z^d \mid x_1 = x_1^{'} \wedge \cdots \wedge x_{l-1} = x_{l-1}^{'} \wedge x_l < x_l^{'}\}.$$

Because $J_l(x)$ represents a parametric polytope, it's cardinal can be efficiently computed [101, 99] by a recent extension added to PolyLib.

Consider now a data-path $dp =< (d_1, P_1), (d_k, P_k) >$. With dp we associate two functions for quantifying the number of tokens that are produced and consumed, respectively, relative to the channels belonging to the data-path.

- The first one is $rank(x, d_k)$ which is the rank function computed for the domain d_k, and gives the number of tokens that have been consumed before iteration x is fired.
- The second function is $read(x, d_k) = rank(M(x), d_1)$ which gives the number of tokens that have been produced inside d_1 before iteration x is fired at the consumer side.

Data-path storage requirements

With these two functions we can define the notion of the storage capacity of a data-path.

Definition 13 *Consider that between two processes A and B of a network there exist n different data-paths $dp_i =< (pd_i, A), (cd_i, B) >$, $i = 1, 2, ..., n$, each data-path being characterized by a function M_i representing the composition of the individual dependency functions corresponding to each channel part of dp_i. The storage capacity of the data-path $t =< (s, A), (d, B) >$ is:*

$$ST(t) = \max_{i=\overline{1,n}} \{ \max_{x' \preceq x, x \in d} \{rank(M_i(x), s) - rank(x, d)\} \} + 1, \tag{8.6}$$

where $rank(M_i(x), s)$ is the read function relative to the domain s, and $rank(x, d)$ is the rank function computed relative to the domain d.

Here we give an intuitive explanation of the formula expressing the storage capacity of a data-path. $T = \max_{x' \preceq x, x \in d}\{rank(M_i(x), s) - rank(x, d)\}$ is the number of tokens that have to be produced by the producer process A relatively to the mapping M_i, to make the firing of the iteration x at the consumer side. By taking the maximum of T relative to all the data-paths that connect A with B we get the absolute number of tokens (or firings) that have to be produced by process A and sent through data-path t to allow the execution of iteration x at the process B. *In other words, the execution of the iteration x from process B implies a number of $ST(x)$ firings of process A.* Thus the storage capacities of a data-path relating two processes depends on the other data paths that relate the same processes. The following theorem results from the discussion:

Theorem 7 *If a network is free of deadlock then for each data-path dp_i, the sum of the sizes of the component channels is greater than, or equal to $ST(dp_i)$.*

In the next two examples we apply the technique for evaluating the necessary condition for deadlock free PN execution.

Example 1: The network presented in this example includes two adjacent data-paths of which storage requirements are sufficient to avoid artificial deadlock.

The network consists of 3 processes PR_1, PR_2 and PR_3, corresponding to the input code function calls PR_1, PR_2 and PR_3, respectively, and 3 FIFOs $FIFO1, FIFO2$ and $FIFO3$. The two adjacent data-paths are: $dp_1 =< (d_1, PR_1), (d_4, PR_2) >$ and $dp_2 =< (d_2, PR_1), (d_4, PR_2) >$, where:

$$\begin{aligned} d_1 &= \{(i, j) \mid 1 \le i \le N, 1 \le j \le N\}, \\ d_2 &= \{(i, j) \mid N \le i \le 2N, 1 \le j \le N\}, \\ d_4 &= \{(i, j) \mid 1 \le i \le N, 1 \le j \le N\}. \end{aligned}$$

The data-path dp_2 consists of two FIFOs $FIFO2$ and $FIFO3$ while dp_1 consists of only $FIFO1$. The mapping of dp_2 is a translation $M(i, j) = M_2 \circ M_3(i, j) = (i + N, j)$, while the mapping of dp_1 is the identity $M_1(i, j) = (i, j)$.

First we compute the storage capacity for dp_1. Following (8.6) we compute the two component rank functions:

$$rank(x, d_1) = \begin{cases} (i-1)N + j, & (1,1) \preceq x \preceq (N, N); \\ N^2, & (N, 1) \preceq x, \end{cases}$$

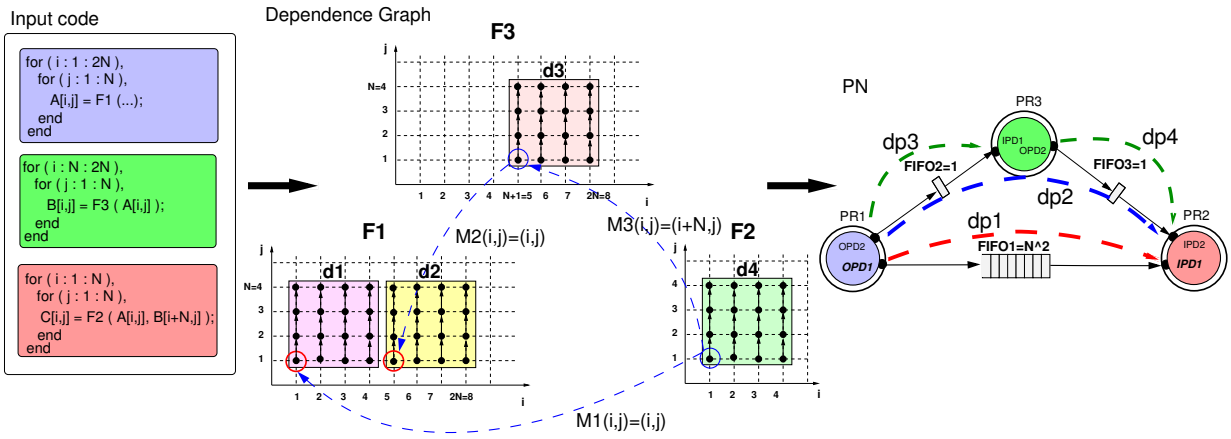


Figure 8.3: PN free of deadlock

and $rank(x, d_4) = (i-1)N + j$, where $x = (i, j)$. Now we are able to compute for mapping M_1:

$$\max_{x \in d_4}\{rank(M_1(x), d_1) - rank(x, d_4)\} = \max_{x \in d_4}\{(i-1)N + j - (i-1)N - j\} = 0. \tag{8.7}$$

Observe that as a consequence of the in-order communication between PR_1 and PR_2, for all $x \in d_4$ the max value is always 0. Next we compute the max expression for the mapping M as: $\max_{x \in d_4}\{rank(M(x), d_1) - rank(x, d_4)\} = \max_{x \in d_4}\{N^2 - (i-1)N - j\} = N^2 - 1$, with the maximal value achieved at iteration $(1, 1) \in d_4$. The maximum of the two maximums is $N^2 - 1$ such that according to (8.6) we conclude that the *minimum size for* $FIFO1$ *is* N^2.

Second we compute the storage capacity for dp_2. This data path consists of two FIFOs: $FIFO2$ and $FIFO3$. Following (8.6) we compute the two component rank functions:

$$rank(x, d_2) = \begin{cases} 0, & x \prec (N+1, 1); \\ (i-N-1)N + j, & (N, 1) \preceq x \preceq (N, N), \end{cases}$$

and $rank(x, d_4) = (i-1)N + j$. Now we are able to compute for the mapping M:

$$\max_{x \in d_4}\{rank(M(x), d_2) - rank(x, d_4)\} = \max_{x \in d_4}\{((i+N) - N - 1)N + j - (i-1)N - j\} = 0. \tag{8.8}$$

Similarly we compute the same *max* expression for mapping M_1 as:

$$\max_{x \in d_4}\{rank(M_1(x), d_2) - rank(x, d_4)\} = \max_{x \in d_4}\{0 - (i-1)N - j\} = -1. \tag{8.9}$$

The maximum of the two maximums given by (8.8) and (8.9) is equal to 0. So we conclude that the *minimum size for the FIFOs* $FIFO1$ *and* $FIFO3$ *is equal to* 1.

From the analysis done so far, in order to avoid deadlock the FIFO channels have the following characteristics:

$$\begin{cases} min(FIFO1) \geq N^2; \\ min(FIFO2 + FIFO3) \geq 1; \end{cases} \tag{8.10}$$

Apart from the adjacent data-paths dp_1 and dp_2 there are two other data-paths, $dp_3 =< (d_2, PR_1), (d_3, PR_3) >$ and $dp_1 =< (d_3, PR_3), (d_4, PR_2) >$. For both of these data-paths the minimum amount of storage capacities to avoid deadlock is:

$$\begin{cases} min(FIFO2) \geq 1; \\ min(FIFO3) \geq 1; \end{cases} \tag{8.11}$$

Therefore to avoid artificial deadlock is necessary that,

$$\begin{cases} min(FIFO1) \geq N^2; \\ min(FIFO2) \geq 1; \\ min(FIFO3) \geq 1; \end{cases} \tag{8.12}$$

By doing a simple analysis you can see that those conditions are also sufficient in order to avoid deadlock. Notice also that while the input sequential program requires for storage $2N^2 + N^2 + N^2 = 4N^2$ memory locations, the final network requires only $2N^2 + 2$ (FIFO) locations.

Example 2: Here we present an example where although for each network data-path, the sum of the sizes of the component channels is greater or equal then $ST(dp_i)$, the network deadlocks. In other words this example shows that the data-path conditions from Theorem 7 are not sufficient to guarantee free of deadlock networks.

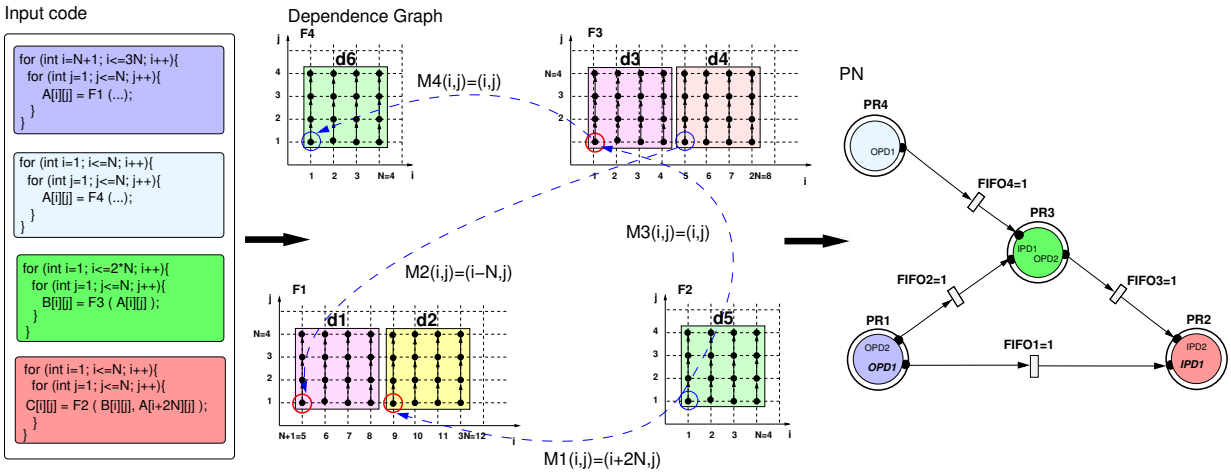


Figure 8.4: Deadlock under the conditions imposed by Theorem 7

Consider the input program given in the left part of Figure 8.4 with it's associated dependence graph and resulted process network. This network does not contain any adjacent data-paths. Moreover, each of the communication channels are in order. Hence, the necessary condition for avoiding deadlock would be to allocated for each channel one memory location: $min(FIFO1) \geq 1, min(FIFO2) \geq 1, min(FIFO3) \geq 1, min(FIFO4) \geq 1$. However, this would lead to deadlock, that could be avoided by allocating to $FIFO1$ or to $FIFO2$ a size equal to N^2: $min(FIFO2) \geq N^2$ or $min(FIFO3) \geq N^2$. The reason behind the deadlock is the existence of a ordering relation between IPs belonging to processes $PR1$ and $PR2$ due to the lexicographic ordering of the iterations belonging to the domains $d3$ and $d4$ part of process $PR3$. The analysis of this kind of behavior is let for further research.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, Austin, Texas, 1983. ACM Press.

[3] D. Avis. lrs: A revised implementation of the reverse search vertex enumeration algorithm. In MG. Kalai and G. Ziegler, editors, *In: Polytopes - Combinatorics and Computation*, pages 177–198. Birkhauser-Verlag, March 2000.

[4] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229, Madrid, Spain, 2002. Springer-Verlag, Berlin.

[5] Uptal Banerjee. Dependence analysis. *The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers*, 1997.

[6] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[7] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[8] 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.

[9] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In *Communicating Process Architectures - 2001, Proceedings*, pages 1–14, Bristol, UK, September 2001.

[10] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1967.

[11] Aart J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[12] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology Journal*, February 2001.

[13] T. Brandes. The importance of direct dependences for automatic parallelization. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 407–417, New York, NY, USA, 1988. ACM Press.

[14] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[15] Joseph Buck and Edward A. Lee. The token flow model. In *Data Flow Workshop*, pages 267–290. Hamilton Island, Australia, May 1992.

[16] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California, Berkeley CA 94720, 1993.

[17] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frederic Vivien. On the removal of anti- and output-dependences. *Int. J. Parallel Program.*, 26(3):285–312, 1998.

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

[19] T. Christof and A. Lobel. Porta: Polyhedron representation transformation algorithm. In *http://www.iwr.uni-heidelberg.de/groups/comopt/software/PORTA/*, 1992.

[20] Ioan Cimpian, Alexandru Turjan, Erwin de Kock, and Ed Deprettere. Communication Optimization in Compaan Process Networks. In *Proceedings of the 4th International Workshop on Systems, Architectures, MOdeling, and Simulation SAMOS'04*, Samos, Greece, July 19-21 2004.

[21] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.

[22] Philippe Clauss and Vincent Loechner. polylib. http://icps.u-strabg.fr/polylib, September 2005.

[23] Jean-Francois Collard, Tanguy Risset, and Paul Feautrier. Construction of do loops from systems of affine constraints. *Parallel Processing Letters*, 5:421–436, 1995.

[24] W. Cook, A. Gerards, A. Schrijver, and E. Tardos. Sensitivity theorems in integer linear programming. In *Mathematical Programming*, pages 251–264, 1986.

[25] G. Dantzig. Programming of interdependent activities, mathematical model. *Econometrica*, pages 200–211, 1949.

[26] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[27] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 298–308, New York, NY, USA, 2003. ACM Press.

[28] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2000.

[29] E De Greef, F Catthoor, and H De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Processing and Multimedia*, Geneva, Switzerland, July 1977.

[30] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.

[31] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.

[32] Giovanni De Micheli. Synthesis and optimization of digital circuits, 1994. McGraw-Hill series in electrical and computer engineering. Electronics and VLSI circuits.

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

[34] Jos van Eijndhoven, Jan Hoogerbrugge, M.N. Jayram, Paul Stravers, and Andrei Terechko. Cache-coherent heterogeneous multiprocessing as basis for streaming application. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices,Philips Research Book Series*, volume 3, pages 61–80. Springer, 2005.

[35] Hesham El-Rewini, Ted Lewis, and Bruce Shriver. Guest editors' introduction: Parallel and distributed systems-from theory to practice. *IEEE Parallel Distrib. Technol.*, 1(3):7–11, 1993.

[36] Paul Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, 1988.

[37] Paul Feautrier. Parametric Integer Programming. In *RAIRO Recherche Oprationnelle, 22(3):243-268*, 1988.

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

[39] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, pages 91–111, 1995.

[40] E. Gawrilow, M. Joswig, and G. Ziegler. Polymake: a framework for analyzing convex polytopes, 1996.

[41] M.C.W. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *In P. Degano, Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, pages 319–334, Warsaw, Poland, 7-11 April 2003 2003.

[42] Kourosh Gharachorloo, Vivek Sarkar, and John L. Hennessy. A simple and efficient implmentation approach for single assignment languages. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 259–268, New York, NY, USA, 1988. ACM Press.

[43] G. Goff, Ken Kennedy, and C-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 15–29, Toronto, Ontario, Canada, June 1991.

[44] R. Gomory. An algorithm for integer linear programming. In Graves and Wolfe, editors, *Recent Advances in Mathematical Programming*, New York, NY, USA, 1963. McGraw-Hill.

[45] Gert Goossens, Dirk Lanneer, Marc Pauwels, Francis Depuydt, Koen Schoofs, Augusli Kifli, Marco Cornero, Paolo Petroni, Francky Catthoor, and Hugo De Man. Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures. *J. VLSI Signal Process. Syst.*, 9(1-2):49–65, 1995.

[46] H. Greenberg. *Integer Programming*. Academic Press, 1971.

[47] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 47, 1995.

[48] Tim Harriss, Richard Walke, Bart Kienhuis, and Ed Deprettere. Compilation from matlab to process networks realized in fpga. *Design automation of Embedded Systems*, 7(4), November 2002.

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

[50] B. Jeannet. Convex polyhedra library - release 1.1.3c edition. In *Documentation of the New Polka library available at http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html*, March 2002.

[51] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., August 5-10 1974.

[52] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. In *SIAM J. Comput., 8 4*, pages 499–507, 1979.

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

[54] Vinod Kathail, Shail Aditya, Robert Schreiber, and Bob Rau. PICO: Automatically Designing Custom Computers. *IEEE Computer*, 35(9), September 2002.

[55] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, page 332, Washington, DC, USA, 1995. IEEE Computer Society.

[56] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

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

[58] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000.

[59] D.J. Kuck. *The Structure od Computers and Computations*, volume 1. John Wiley and Sons, New York, NY, USA, 1978.

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

[61] Edward Lee et al. Ptolemyll: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[62] H. LeVerge. A note on Chernikova's algorithm, Research Report 635, IRISA, February 1992.

[63] D.E. Maydan, S.P. Amarashinghe, and M.S. Lam. Data Dependence and Data-Flow Analysis of Arrays. In *Proc. of 5th Workshop of Languages and Compilers for Parallel Computing*, pages 434–448, August 1992.

[64] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[65] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1991. ACM Press.

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

[67] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-interscience, 1988.

[68] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Efficient automated synthesis, programming, and implementation of multi-processor platforms on fpga chips. In *16th Int. Conference on Field Programmable Logic and Applications (FPL'06)*, pages 323–328, 2006.

[69] 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. ACM Press, 2006.

[70] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[71] Tom Parks. Bounded scheduling of process networks, December 1995. T. M. Parks, Bounded Scheduling of Process Networks, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California, Berkeley.

[72] http://www.picochip.com.

[73] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[74] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[75] William Pugh and David Wonnacott. An Exact Method for Analysis of Value-based Array Data Dependences. In *Proceedings of the Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, Dec 93.

[76] Fabien Quillere and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. In *ACM Transactions on Programming Languages and Systems*, volume 22, pages 773–815, September 2000.

[77] Fabien Quillere, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal Parallel Program.*, 28(5):469–498, 2000.

[78] Michael J. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[79] Edwin Rijpkema. *Modeling Task Level Parallelism in Piece-wise Regular Programs*. PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, September 2002.

[80] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4–17, January 2005.

[81] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.

[82] Todor Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, September 2004.

[83] Todor Stefanov, Bart Kienhuis, and Ed Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proceedings of 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, may 6–8 2002.

[84] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. "7th Int. Conf. Design, Automation and Test in Europe (DATE'04)"*, Paris, France, Feb 16 – 20 2004.

[85] Paul Stravers and Jan Hoogerbrugge. Homogeneous multiprocessors and the future of silicon design paradigms. In *Proceedings of the Int. Symposium on VLSI Technology, Systems, and Applications*, April 2001.

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

[87] Ying Tao. Internal technical report 06-21, heterogeneous multiprocessor system design with espam: Integration of hardware ip core. Master's thesis, LIACS - Leiden University, The Netherlands, December 2006.

[88] Robert Endre Tarjan. Finding a maximum clique. *Department of Computer Science, Cornell University*, TR 72-123, 1972.

[89] Lothar Thiele. On the Design of Piecewise Regular Processor Arrays. In *Proc. IEEE Symp. on Circuits and Systems*, pages 2239–2242, Portland, USA, 1989.

[90] X. Tian, A.J.C. Bik, M. Girkar, P.M. Grey, H. Saito, and E. Su. Intel openmp c++/fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6, February 2002.

[91] 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.

[92] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. *Realizations of the Extended Linearization Model*. Marcel Dekker, Inc., Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation edition, 2003.

[93] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A hierarchical classification scheme to derive interprocess communication in process networks. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[94] 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.

[95] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating affine nested loops to Process Networks. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Washington D.C., September 23–25 2004.

[96] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Classifying interprocess communication in process network representation of nested loop programs. In *ACM Transactions on Embedded Computing Systems*, Expected 2007.

[97] Alexandru Turjan, Todor Stefanov, Bart Kienhuis, and Ed F. Deprettere. The COMPAAN Tool Chain to convert Matlab into Process Networks - Software DEMO. In *Designers' Forum at Design, Automation and Test in Europe (DATE'2002)*, pages 258–259, Paris, France, March 4-8 2002.

[98] V Vanhoof, I Bolsens, and Hugo De Man. Compiling multi-dimensional data streams into distributed dsp asic memory. In *In Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara, CA, August 1989.

[99] Sven Verdoolaege. *Incremental loop transformations and enumeration of parametric sets*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, April 2005.

[100] Sven Verdoolaege, Kristof Beyis, Maurice Bruynooghe, Rachid Seghir, and Vincent Loechner. Analytical Computation of Ehrhart Polynomials and its Applications for Embedded Systems. In *2nd Workshop on Optimizations for DSP and Embedded Systems, (ODES'04)*, Palo Alto, CA, 2004.

[101] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–258, New York, NY, USA, 2004. ACM Press.

[102] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, and J. L. van Meerbergen. Multidimensional periodic scheduling: a solution approach. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 468, Washington, DC, USA, 1997. IEEE Computer Society.

[103] Wim F. J. Verhaegh, Emile H. L. Aarts, and Paul C. N. van Gorp. Period assignment in multidimensional periodic scheduling. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 585–592, New York, NY, USA, 1998. ACM Press.

[104] D. K. Wilde and Sanjay Rajopadhye. Allocating memory arrays for polyhedra. Technical Report RR-2059.

[105] Doran Wilde. A library for doing polyhedral operations. In *Technical Report PI 785, IRISA, Rennes, France*, 1993.

[106] Doran Wilde and Sanjay Rajopadhye. Memory reuse in the polyhedral model. In *In Proc. Euro-Par96*, Lyon, France, August 2002.

[107] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[108] http://www.xilinx.com.

[109] Claudiu Zissulescu. *Synthesizing Process Networks to VHDL*. PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, Expected 2007.

[110] Claudiu Zissulescu and Bart Kienhuis. Communication Synthesis in a Multiprocessor Environment. In *Proc. 15th Int. Conference on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, 2005.

[111] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Increasing pipelined ip core utilization in process networks using exploration. In *Proc. 14th Int. Conference on Field Programmable Logic and Applications (FPL'04)*, Antwerp, Belgium, 2004.

[112] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Expression synthesis in process networks generated by laura. In *Proceedings of the IEEE 16th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'05)*, Samos, Greece, 2005.

[113] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, 2003.

Index

adjacent data-paths, 127
array data flow analysis, 4

Basic FIFO Merging Test (BFMT), 118

Communication Model Selection, 7, 41
Compaan compiler, 2
Consumption Restructuring, 6
content addressable memory (CAM), 67

data dependence, 4
Data Dependence Graph (DDG), 4
data-path, 127
deadlock, 127
Dual Multiplicity Problem, 44

Empty Domain Test, 16
execution order, 12
explicit dependency function, 88

FIFO.get(), 115
FIFO.put(), 115
Fourier-Motzkin elimination, 16

General FIFO Merging Test (GFMT), 118
get, 95
Gomory Cuts, 16

Hermite Normal Form (HNF), 14
heterogeneous multiprocessor platforms, 1

if-conversion, 125
in-order matrix, 55
in-order with multiplicity (IOM+), 43
in-order without multiplicity (IOM-), 43
Input Port Domain (IPD), 17
integer linear programming (ILP), 6
irregular hole, 33
iteration domain, 12
iteration points (IPs), 6
iteration space (IS), 6, 12

Kahn Process Network (KPN), 67

lattice maximal function, 89
Laura, 8
Lexicographic Dual Simplex, 16
lexicographic order, 12
Lexicographically Maximal Preimage (LMP), 75
Lexicographically Minimal Preimage (LmP), 75
lifetime analysis, 74
lineality space, 58
linearly bounded lattice (LBL), 24

model of computation (MoC), 1
multiplicity, 42
Multiplicity Problem, 43
Multiplicity Test, 43

objective, 15
Omega test, 21
out-of-order with multiplicity (OOM+), 43
out-of-order without multiplicity (OOM-), 43
Output Port Domain (OPD), 23

Panda, 123
PNGen, 123
Polyhedral Model, 11
polyhedral model, 6
Presburger formulas, 21
Primal Multiplicity Problem, 44
Process Network (PN), 1
Process Network Generator, 123
Process Network Synthesis, 8
Process Splitting, 104, 105
producer expanded domain, 86
producer expansion function, 85
producer/consumer (P/C) pair, 7
Production Restructuring, 7
pseudo multidimensional knapsack, 44
put, 95

random addressable memory (RAM), 67

read function, 69
Reordering Problem, 43, 49
Reordering Test, 43, 49

sequential program, 67
Smith Normal Form (SNF), 13
static affine nested loop programs (SANLP), 11

Theorem of Dependence, 72
total order, 115

validity bit, 72

write function, 69

Acknowledgments

I would like to thank the following fellow Ph.D. students for the discussions we had: Claudiu Zissulescu, Todor Stefanov, Vladimir Zivkovic, Laurentiu Nicolae, Ioan Cimpian, Hristo Nikolov, Mihai Cristea and Sjoerd Meijer. I also want to thank Kapil Makhija and Gideon Liem for the collaboration we had during their student projects.

In the context of my research project I would like to thank the following researchers for the co-operation we had: Claudiu Zissulescu, Todor Stefanov, Vladimir Zivkovic, Ioan Cimpian, Alain Darte, Erwin de Kock, Tanguy Risset, Paul Feautrier, Steven Derrien, Patrice Quinton and Shuvra Bhattacharyya. In particular, I want to thank Dmitry Cheresiz for reading and giving me useful suggestions regarding my papers, and Alain Darte for inviting me to Lyon and carefully reading one of my papers. I would like also to thank Paul Feautrier for his open source PIP library.

I am also grateful to the following people for reading my thesis manuscript and providing me very useful feedback: Erwin de Kock, Dmitry Cheresiz, Sjoerd Meijer, Sven Verdoolaege and Sebastian Pop.

I also want to thank Eugenie Baken for being a close friend and helping me to settle in The Netherlands, and Stefka and Todor Stefanov for sharing their house with me.

Several of my professors have contributed for building my way of thinking and knowledge at a level which gave me the confidence to start a Ph.D. research. In particular, I want to thank Bogdan Mitu, Gheorghe Stefan, Ovidiu Bogdan, Ionel Stanica and Gheorghe Mateescu.

Finally, I would like to thank Maaike and Dmitry for supporting me through the years.

Samenvatting

Applicaties uit het veld van de multi-media, bio-informatica, en signaalverwerking zullen steeds meer rekenkracht vereisen. Om aan deze vraag tegemoet te komen verschijnen er steeds meer computerarchitecturen, waaronder de heterogene multi-processor platforms. Deze bestaan uit componenten waar data uitgewisseld wordt door gebruik te maken van programmeerbare interconnections, een switch matrix, of een network on chip. Bij de componenten zelf kan gedacht worden aan CPUs, DSPs, IP cores, herconfigureerbare units, en geheugens.

Ondanks dat zulke heterogene systemen al gefabriceerd worden, ligt de verantwoordelijkheid van het mappen van de applicaties bij de systeemontwerper en zijn deskundigheid om het geheugen en aansturing van de applicatie handmatig over de componenten te verdelen. Dit wordt typisch op een empirische manier uitgevoerd vanwege het ontbreken van een systematische aanpak. Tijdens dit proces gaat de aandacht van de ontwerper voornamelijk uit naar het herkennen van onafhankelijke taken binnen de applicatie, de synchronizatie hiervan, en het geheugenbeheer.

Wij geloven dat het werk verricht in homogene multi-processor systemen niet de oplossing zal zijn voor het programmeren van heterogene systemen en applicaties waarin wij geinteresseerd zijn. Bij compilatietechnieken voor multi-processor systemen gaat de meeste aandacht uit naar systemen die een geheugen delen (shared memory model) en waar het werk verdeeld moet worden over een aantal processors van hetzelfde type. Hierbij is een lineaire snelheidsverbetering de belangrijkste maatstaaf. Bij heterogene multi-processor systemen gaat het echter om het verdelen van het werk over verschillende componenten. Het doel is om bij soort systemen een voordeel te halen uit de heterogeniteit van het systeem en zo de beste trade-off te krijgen van flexibiliteit en snelheid. Wij geloven dat het Process Netwerk (PN) rekenmodel zeer geschikt is om heterogene multi-processor systemen te programmeren. Het PN is een deterministisch rekenmodel waarin taken als processen gedefinieerd worden en gedistribueerd geheugen als FIFO kanalen. De lokale aansturing van de applicatie en gedistribueerde geheugens zijn de belangrijkste elementen van het PN model die het programmeren van heterogene systemen toestaan. De lokale aansturing komt goed overeen met de verschillende componenten van het systeem en de gedistribueerde geheugens met de stijl waarin de verschillende componenten communiceren. Het schrijven van applicaties in de vorm van het PN model is echter zeer tijdrovend en foutgevoelig. Om deze redenen hebben wij de compaan compiler ontwikkeld die affine nested-loop programma's vertaald naar functioneel equivalente PNs in C++, Java, of VHDL.

In deze thesis worden compilatietechnieken gepresenteerd die het vertalen van een complete klasse van statische affine nested-loop programma's naar een multi-threaded respresentatie in de vorm van een process netwerk mogelijk maakt. In onze aanpak wordt de gebruiker bevrijd van de taak om zich bezig te houden met de details van het partitioneren van iteratieruimte, het delen van data, en thread synchronizatie. Onze aanpak is analystisch. Hiermee bedoelen we dat er geen enkele

heuristiek gebruikt wordt. De aanpak kan onderverdeeld worden in vier stappen die we elk in een apart hoofdstuk behandelen. Verder laten we zien hoe elke stap geformuleerd kan worden als een Integer Linear Programming (ILP) probleem en hoe dat opgelost kan worden door gebruik te maken van ILP of andere minder zware rekenkundige technieken. Bovendien illustreren we elke stap met een voorbeeld.