Internal Report 2012–06 June 2012

Universiteit Leiden
Opleiding Informatica

Increasing Explicit Parallelism
in Polyhedral Process Networks
using Transformations.

Wouter de Zwijger

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Increasing Explicit Parallelism in Polyhedral Process Networks using Transformations.

M.Sc. thesis of

Wouter de Zwijger

Supervisors:
drs. Sven van Haastregt
dr. ir. Todor Stefanov
Second reader:
dr. Jetty Kleijn

Student ID: 0611735

29 June, 2012

LIACS, Leiden University

Abstract

Design of modern embedded systems is becoming increasingly complex because of increasing system requirements. Design tools help designers to cope with the increasing complexity. One of these design tools is the polyhedral process network based Daedalus tool flow, which allows a designer to derive implementations starting from a high level C code representation. So far, the tool flow did not provide means to automatically control the degree of exploited parallelism. We have extended the Daedalus tool flow to allow the designer to transform the application on a high level. Our method enables the designer to increase the performance of a system without the need to change the application specification itself. We also propose a method to analyze the high level structure of the application to automatically determine transformation parameters. The designer can then either use this solution or use it as guide to determine a satisfactory system configuration.

Contents

Chapter 1

Introduction

1.1 Embedded Systems And Design Tools

Embedded systems [1] are systems designed for a specific purpose. They have close interaction with the environment they are in. Embedded systems have sensors to detect changes in the environment. This information is processed by the embedded system and can be followed by a response from the embedded system to the environment. It is important that the response of the embedded system is in time, as the environmental conditions are constantly changing.

A lot of electronic devices are embedded systems. A few examples of embedded systems are those in video recorders, radar systems, washing machines and televisions. All of these objects contain processors which have a certain goal to achieve. Since these systems should perform a specific task, they do not need to execute this as fast as possible, just fast enough. The processors in a television must convert the signal it receives by cable to images on its screen. Processing the incoming data should not necessarily be processed as fast as possible, but fast enough to keep up with the incoming data. Processing the data too slow might result in a unsatisfying user experience, as the animations on the tv screen are not presented in the intended speed. Processing the data in time will also prevent the data to heap up. Since the tv has finite buffers, data heaped up should at some point result in lost of data, giving an unsatisfying user experience as the observed animations of the screen skip some frames.

Designing the processors of an embedded system is a complex process. Since the implementation is dependent on specialized hardware, it is not possible to test them in practice early in the

design phase. FPGAs (Field-Programmable Gate Arrays) are used for testing and development of these systems. The logical gates from these chips can be configured to implement different designs.

Technology increases quickly and embedded Systems are used for more complex tasks which require more processing power. The current trend is to improve the processing power by putting multiple processors on one chip. These processors can execute in parallel, which increases the performance. The processors can not execute in parallel if they are dependent on each other by data. When a processor requires data which is a result from another processor, it needs to wait until this data is processed by the other processor. The increasing number of processors make it more complex to keep track of these dependencies and thus designing and testing embedded systems.

With the assistance of design automation tools designing embedded systems can be done on higher levels. The designer no longer needs to take care of the exact implementation, but rather what the embedded system should accomplish. Instead of specifying the functionality of an embedded system by defining the exact hardware configuration, designers can write the functionality in a higher language, for example in C. To translate the higher level definition of the embedded system into its exact lower level implementation, tools are required. These tools use the input of the designer and automatically translate it in the actual hardware configuration.

Implementing systems on a low level means that it can not be reused in other projects later. With the rising costs of development of these complicated systems it would be very beneficial to reuse parts. That is why platform based tools are an outcome [2].

Using different tools developers can design systems for MP-SoCs (MultiProcessor System on Chips). Bridging the gap between designed models to actual prototypes and simulation another set of tools is required. One such tool is Daedalus.

Daedalus[3] is an open-source design flow for MP-SoCs. It can be used for Design Space Exploration (DSE). The flow of Daedalus starts from C code and generates hardware specifications of simulation of the C program or mapping on FPGAs. The tool flow is highly automatic.

1.2 Problem Statement

The Daedalus tool flow is capable of translating an embedded system design of higher level specification to implementation, either for simulation or implementation on hardware.

The designer might want to specify how much hardware resources the designed system could use, and explore the impact on performance when the amount of hardware resources changes.

The current Daedalus tool set already support this. However, increasing the number of hardware resources often does not yield performance increase, unless the amount of tasks that can execute in parallel in the application matches the amount of processing resources. The designer could try to influence the number of tasks in the application by applying tricks on the C code defining the system's functionality, but this is tedious and potentially change the programs functionality when done incorrectly. It is also not very transparent how changing the C code affects the final hardware implementation. Because the tools of Daedalus do not support the manipulation of explicit parallelism in an application, it is also not possible for the design flow to automatically determine the most satisfying hardware configuration for the designer. The tool flow makes sure sufficient hardware is allocated to perform the required task, but does not take into account that the application might need to be optimized to perform the task faster. To improve the design process and allowing the designer the change the amount of hardware resources, the tools should support a way to manipulate the amount of hardware resources used and the ability to determine hardware configurations for given performance criteria.

In this thesis we attempt to tackle the current limitations of the Daedalus tool flow by developing a number of methods to manipulate and optimize the amount of tasks allocated by the design flow. The problem is split in two subproblems;

1. How to manipulate the application specification such that the amount of explicit parallelism will be increased as desired while preserving functional behavior.

2. Which manipulations of the application specification lead to the desired system performance without wasting unnecessary hardware resources.

The first part of the problem is to find a method to modify the hardware resources configuration of the implementation. With this method the designer could experiment with different implementations of the same system by allocating different amounts of hardware resources. The designer can use the findings to adjust its final system to allocate sufficient hardware to do an intended task.

The second part of the problem is to automate the process of finding a satisfying hardware configuration for a given system specification and performance criteria. If this can be done automatically this will save the designer a lot of time. It is also possible that the automatic analysis method will find a better solution the designer could find on its own. The designer could also use the solution of the analysis method as a guide to find an optimal solution for the designer's needs.

1.3 Related Work

Different solutions exist for handling the growing complexity of embedded systems. We discuss a number of existing platforms for designing embedded systems, and how they allow the designer to influence the hardware implementation.

The System-on-Chip Environment is a design environment for designing embedded systems on heterogeneous target platforms [4]. The environment requires a specification of the design functionality from the designer. With this specification the environment can explore the design space. Based on the outcome of the design space the designer can specify design decisions for the environment to use. The exploration and refinement of design decisions can be repeated until the designer is satisfied, and continues with the rest of the tool flow of the environment.

The feedback the designer gets during the exploration of the design decisions is based on analysis. These analysis are created by simulating the given design specifications. The designer does not change directly how the hardware changes during the space exploration, but specify certain decisions which are used to automatically change the hardware specifications.

SystemCoDesigner[5][6] is a tool flow using a SystemC model to specify the application behavior. Modules are represented as final state machines. The design space exploration phase of SystemCoDesigner requires the SystemC model as well as an architecture template and user requirements. This template contains all hardware components and accelerators that can be used in the exploration phase. A multi-objective optimization algorithm[7] in combination with symbolic optimization techniques are used to search the design space for the best solutions. The designer can then select between the found results the solution it suits best.

MAPS[8] is a framework for MPSoC specialized in parallelizing C code. The framework requires C code and a description of the target implementation. The MAPS framework will analyze this input to make the C code parallel with threads. The analyze process of MAPS consists of a static part, performing compiler like optimizations using intermediate representation. The analyze process also consists of a dynamic part, which tracks all the memory accesses performed to create a view of all data dependencies. The back end of the MAPS framework is the TCT compiler. Which maps the code on processing elements.

Transformations to achieve better performance has been done in [10]. A number of transformations are supported, including unfolding, skewing, plane splitting, and loop transformations. The transformations can be applied on the program code to achieve different hardware configurations. This allows for quick exploration of different configurations. In our approach we differ

from this one in that we do not apply any transformations on the code. Instead we will apply our transformations one level lower, namely on the Polyhedral Dependence Graphs, which are derived from the source code. One of the advantages of our approach is that the source code stays intact. Another advantage is that transformations can be applied independently of the programming language used in the source code. Furthermore, transformations can be applied without knowledge of the used compiler optimization techniques. This makes our solution robust, and not more complicated than necessary. Transforming on this level also allows us to examine possible effects of the transformations with models. This allows us to explore multiple transformation possibilities without actually simulating the application, which can take more time.

1.4 Solution Approach and Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we define a number of terms and definitions used throughout the rest of this thesis.

The first subproblem, manipulating the application specifications such that the amount of explicit parallelism can be increased, is solved in Chapter 3. The Daedalus tool flow transforms C code to PDGs, which are transformed to PPNs. These are used for the eventual implementation. By creating a method which changes the PDG configuration, the resulting PPN changes. This will result in a change of hardware configuration at implementation. The way we transform the PPNs allows for hardware configuration changes without disturbing the functionality.

The second subproblem, how automatically can be determined which manipulations reach to a desired hardware configuration, will be discussed in Chapter 4. Using throughput calculations we determine what parts of the system can be changed for better performance. Using an iterative flow between manipulation and analysis of the system, it is possible to improve the solution for better performance. The analysis method can be guided by adjusting the properties of the parts of the system that represent the communication with outside the system.

In Chapter 5 we demonstrate the designed methods of Chapters 3 and 4 by a number of examples. We discuss the effects of different ways to transform the PPN and how well the automatic performance improvement method performs. We end the chapter with a number of limitations to the presented methods.

Chapter 6 discusses the work that is done in the previous chapters, and reflects how the problem stated in Chapter 1 is dealt with. Chapter 7 suggest what work should be done in the future to improve the current state of the methods to deal with the problem of designing embedded systems.

An added Appendix shows in detail how the implemented tools designed with the methods of Chapters 3 and 4 can be used. This chapter mainly includes commands that the implemented tools can receive as input.

Chapter 2

Preliminaries

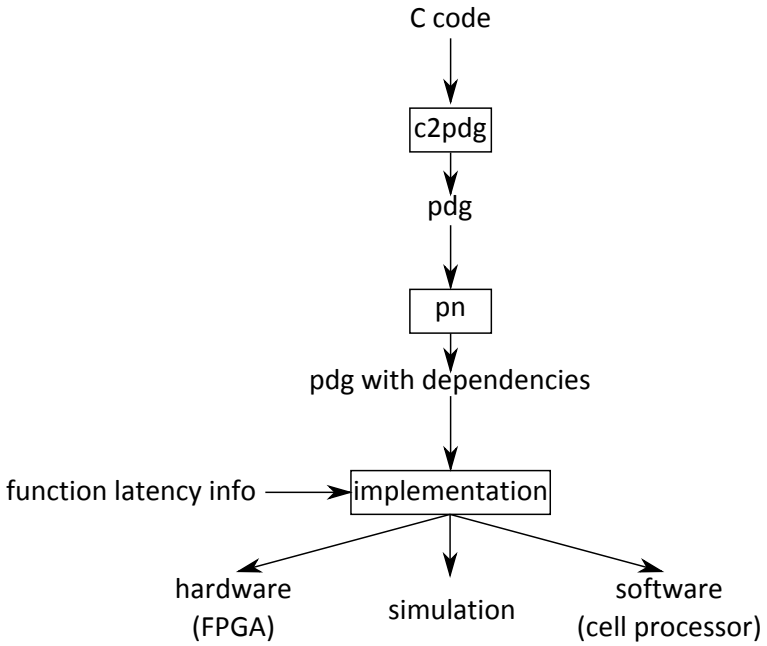

Figure 2.1: Tool flow, showing how C code is converted to PDGs, dependencies are added, and eventually used for implementation purposes.

2.1 Tool Flow

The Daedalus tool flow requires a static affine nested loop program and a number of implementation properties of this program. With this information the Daedalus tool flow can compute a simulation of the program on hardware resources or specifications for actual hardware implementation. The Daedalus tool flow is depicted in Figure 2.1. The C code the designer specifies is translated in a number of representations before it results in a simulation, hardware implementation, or implementation instructions for an FPGA. The C code is first converted with the `c2pdg` tool in a Polyhedral Dependence Graph (PDG). The `pn` tool adds dependencies to the PDG. Note that the PDG already had the information about the dependencies in the previous step of the tool flow, but the `pn` tool makes these explicit. In the implementation step of the Daedalus tool flow, the PDG with dependencies is converted into a Polyhedral Process Network (PPN). With specified function latency info and the PPN, the Daedalus tool flow can create a hardware simulation of the program or actually implement the program on hardware.

In the remainder of this chapter we will explain a number of definitions, specifying the details of PPNs, PDGs, their correlation and, the correlation between them and the C code specified by a designer.

2.2 Sets and Domains

A set is a collection of elements. Each element is unique in the set. An element in a set can be a vector. A vector consists of an ordered list of values. A set has a lexicographical ordering. This means that for each element of the set it can be determined if this element comes before or after another element of that set. Comparing two vector elements of a set, the vector whose first value is the lowest is before the other vector in the lexicographical ordering. If the first value of both vectors is equal, the next value of the vectors is compared to determine which vector is before the other. If this also is the same value, the value after that is compared, etcetera. For example, vector $a = (1, 2, 5)$ and vector $b = (1, 4, 2)$ are both in the same set. Since the first value of both vectors is 1, the next value should be taken into evaluation. The second value of a is smaller than the second value of vector b. Therefore, a comes before b in the lexicographical ordering of the set. There are no elements before the element that is first in a set. Likewise, there are no elements after the last element of a set. In this thesis we use the terms before, after, fist and last to indicate the lexicographical ordering of elements in a set. An integer set is a set containing vectors, where

the values of the vectors are integers.

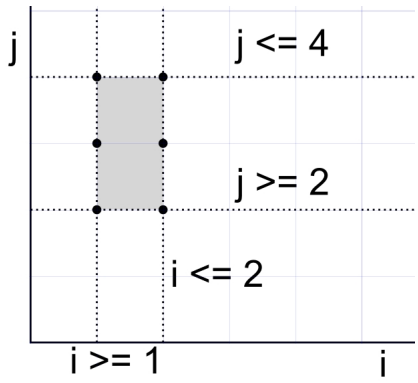

Figure 2.2: An integer set described with inequalities and equalities

Integer sets can be described by equalities and inequalities. For instance a set containing the vectors $(1,2),(1,3),(1,4),(2,2),(2,3),(2,4)$, can be described as $\{[i,j] : i >= 1\ and\ i <= 2\ and\ j >= 2\ and\ j <= 4\}$. This set is a polyhedron as it is described by inequalities and equalities, and is visualized in Figure 2.2.

A set a is a convex set if all integral vectors in the convex hull of a are also in a.

In this thesis we will often use integer sets. Where no confusion can be arise, we will simply use set when we mean integer set. We will use integer sets to describe domains. A domain is a space consisting of multiple dimensions. A domain contains points. A point has a value for each of the dimensions of the domain space. A domain can be expressed as an integer set. In this case each vector of the integer set represents a point of the domain.

2.3 Polyhedral Process Networks

A Polyhedral Process Network (PPN) [11] is a representation of a program. But rather than statements and function calls, a PPN consists of processes and channels between processes as depicted in Figure 2.3. The ellipses represent processes, the arrows in between represent channels.

2.3.1 Process

A PPN contains a number of processes. Each process contains a function that it can execute. A process is in the implementation phase of the Daedalus tool flow mapped on a processor. By

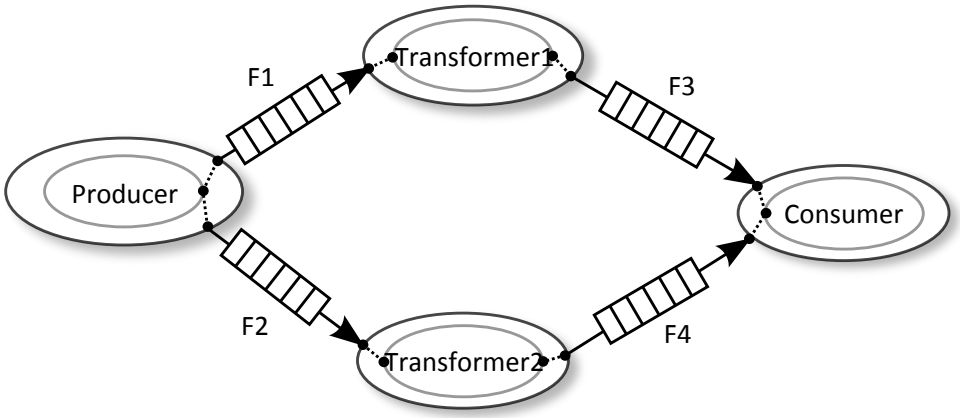

Figure 2.3: Representation of the PPN of Figure 2.5 with the arguments visualized.

default, each process is mapped on a different processor. Processes can thus run concurrently if they are not dependent on each other. Each process has an iteration domain, a set of input arguments and a set of output arguments as explained in the next sub sections.

2.3.2 Iteration Domain

The function of a process executes multiple times. A process representing this function has an iteration domain specifying how often the function must be executed. The iteration domain contains iteration points. Such iteration point or iteration for short is a vector. The values of the vector specify the iteration values of the loops enclosing the original statement.

2.3.3 Arguments

A process has one or more arguments. There are input and output arguments. A process receives data from input arguments, and sends data to output arguments. A process only fires if all input arguments are available. When a process performs a read on an input argument, but there is no input data, the process is blocked, and waits until it receives the required data. If a process tries to write data to an argument, but the FIFO connected to this argument is full, the process is blocked and waits until it can write data. In Figure 2.3, the dots on the inner ellipse of each process represents its arguments. Arguments are connected to channels via ports, and an argument can be connected to multiple channels. In case of an output argument, the data written to the argument is

divided over the connected channels.

2.3.4 Channels

Arguments of processes are connected to each other by channels. A channel is connected to one input argument which reads data from the channel and one output argument which writes data to the channel. The arguments can belong to the same process, or can be from different processes. A channel can be of different types. The most common type for streaming applications is the First-In First-Out (FIFO) channel. This channel has a limited capacity, and thus can get full. A process can not write to an output argument that is connected to a FIFO channel that is full. Data sent through a channel can be represent as data tokens. Each time a process writes to one of its output arguments, a data token will be transported over one of the channels connected to this output argument, and will be received by the input argument from the process connected to the other end of the channel. Data tokens stay in the same order when travelling through a FIFO channel. The token that goes as first into a channel will also be the one that first comes out at the other end.

2.3.5 Static Affine Nested Loop Program

In a Polyhedral Process Network (PPN), [11] statements from the original program are expressed by individual processes. The program needs to be a Static Affine Nested-Loop Program (SANLP) [12] if it is to be converted into a PPN, and thus a program needs to be a SANLP to be converted into a PDG. SANLP are programs with syntax restrictions. The restrictions allow analysis algorithms to determine which statements change the values of arrays, and in what order. These programs do not have while loops and goto statements. A typical SANLP consists of for loops containing statements of variable or array assignments. The bounds of a for loop and array access indices and if conditions are either fixed, determined by an outer loop or dependent on symbolic parameters of the program. Daedalus has been developed as a tool for streaming applications. These applications can often easily being converted into SANLP. The SANLP limitation is therefore mostly not a problem.

2.3.6 Correlation between Polyhedral Process Network and Program Code

Figure 2.4 shows a SANLP. The program consists of functions that are called a fixed number of times by loop constructions. The actual implementation of the functions is omitted, as it is not

```
int main() {
  int a[50];
  int i;

  for (i = 0; i < 50; i++) {
     Producer(&a[i]);
  }
  for (i = 0; i < 25; i++) {
     Transformer1(a[i], &a[i]);
  }
  for (i = 25; i < 50; i++) {
     Transformer2(a[i], &a[i]);
  }
  for (i = 0; i < 50; i++) {
     Consumer(a[i]);
  }

  return 0;
}
```

Figure 2.4: a static affine nested loop program

relevant for the example. In Figure 2.5, the associated PPN is visualized as a graph. Each node of the graph represents a process. Each arrow between two nodes represents a channel between the associated processes. The PPN has four processes: Producer, Transformer1, Transformer2 and Consumer, and four channels F1,F2,F3 and F4. The statements of the SANLP are converted in these processes. The data dependencies between the processes are channels in the PPN. The characters `R`,`E`,`W` indicate the behavior of a process. `R` indicates a statement which reads data from input channels. `E` indicates a statement of a process which computes data. `W` indicates a statement of a process which writes data to output channels.

Transformer1 and Transformer2 both require input from Producer, and Consumer requires input from Transformer1 and Transformer2. A process runs an equal amount of iterations as the associated function executes in the SANLP. Each iteration, the process reads data from incoming channels, executes the function and then writes to its output channels. The producer in Figure 2.5 has only output channels. It starts each iteration with executing the Producer function, followed by writing to one of its two output channels. Transformer1 reads each iteration from channel F1, processes this data, and outputs its result to F3.

In Figure 2.3, the channels between the processes are visualized. The gray ellipses within the nodes indicate the execution functions that are called. The black dots represent input and output arguments. They are the argument values used by the functions of the processes. F3 and F4

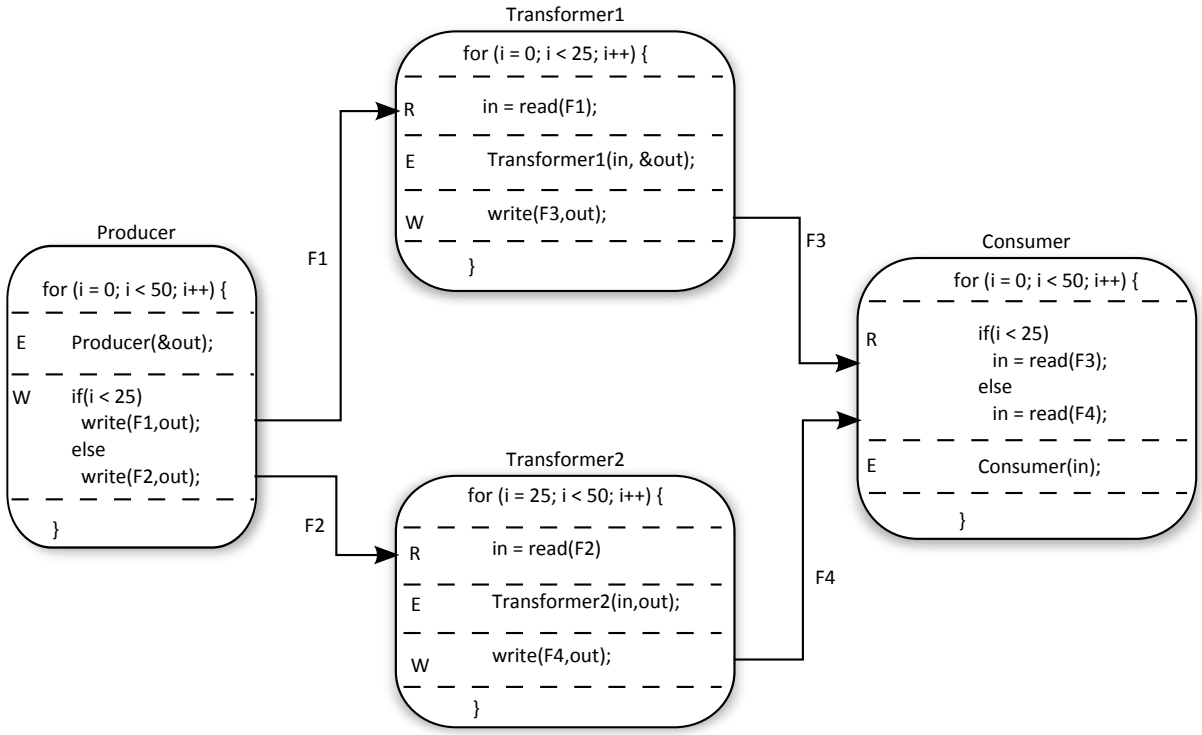

Figure 2.5: The SANLP from Figure 2.4 represented as a PPN.

are both input channels of the Consumer process. They are used in a mutual exclusive way for the same argument for the function of Consumer. The output argument of the Producer process is connected to both F1 and F2. Both channels use the same output argument of the Producer process. The producer process in Figure 2.6(a) has two different output arguments, and so two output arguments. One for each output channel, as visible in Figure 2.6(b).

2.4 Polyhedral Dependence Graphs

In the previous section, we have described the specifications of a PPN. A Polyhedral Dependence Graph is the intermediate representation in the derivation of a PPN. In Table 2.1, the different components of both representations are correlated. Where a PPN consists of processes connected by channels, a PDG consists of nodes dependent on each other through dependencies. PDGs are converted to PPNs for use in the implementation layers of the tool flow.

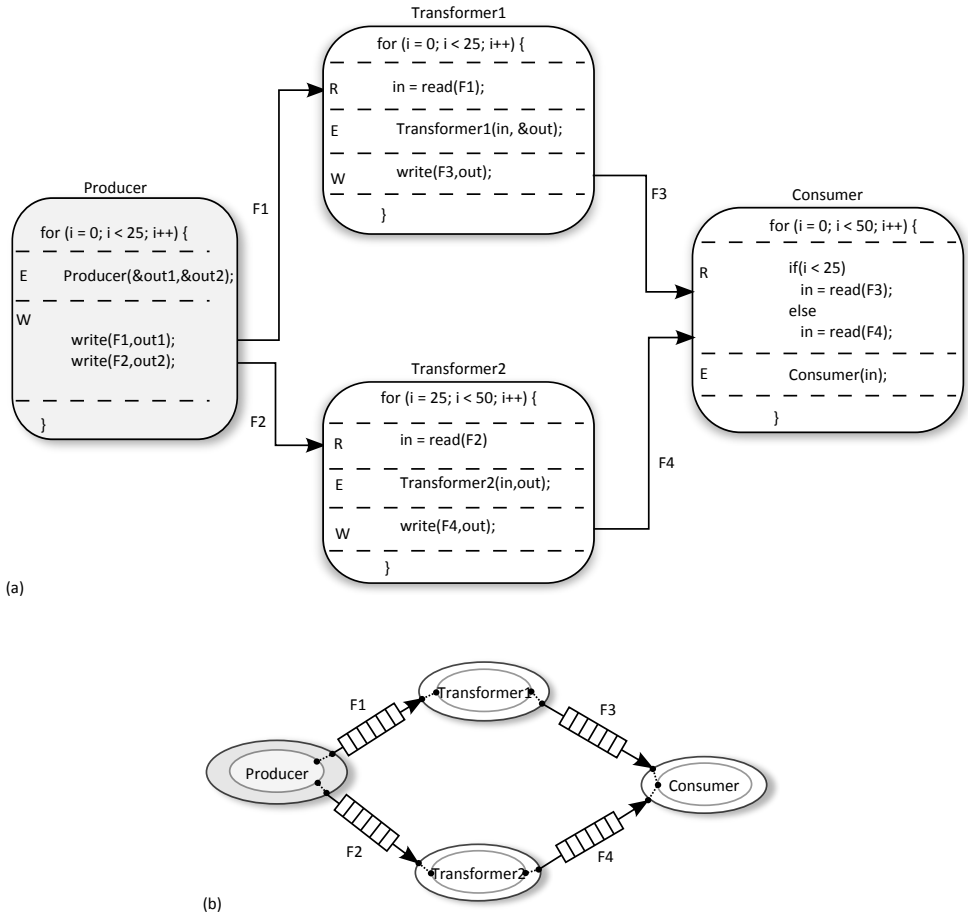

Figure 2.6: Modified PPN from Figure 2.5 and Figure 2.3; the Producer has two output arguments instead of one.

In Figure 2.7, the associated PDG of the SANLP of Figure 2.4 is visualized. The processes of the PPN are nodes in the PDG. Channels between processes in the PPN are dependencies between nodes in the PDG.

The PDG contains the relations between the nodes in the form of polyhedral dependencies. In Figure 2.8, we show this in more detail. Nodes are connected to each other by relations. A node has a set of elements called the source set. The source set contains all iteration points which are the iteration values its function should execute. Iteration points describe the loops that are around the program statement. The iteration points will eventually be simulated in lexicographical order by a scheduler that is part of the Daedalus tool flow, corresponding with the original loops which

PDG	PPN
node	process
dependency	channel
domain of node	domain of node
set element	iteration (point)
statement	function
access field	argument

Table 2.1: Comparison between PPN and PDG terminology.

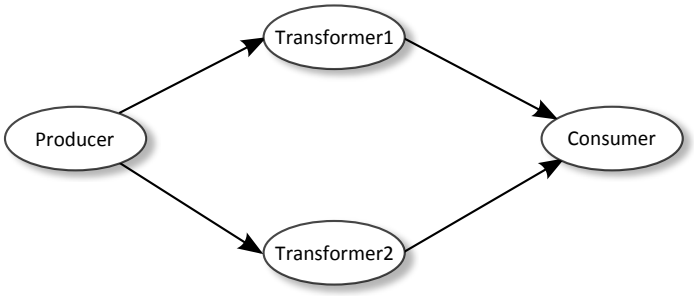

Figure 2.7: Representation as a PDG of the program of Figure 2.4.

are also executed in order. A statement in multiple loops results in a process with a source set containing multi-dimensional vectors.

In Figure 2.8, the detailed information of two of the processes from Figure 2.5 is visualized. The source set of the Transformer2 node is `{ S_2[i] : i >= 25 and i <= 49 }`, consisting of source elements 25 up to and including 49. Note that S_2 is a name space notation, and can be ignored in this thesis. The loop in the original statement Transformer2 came from iterate from 25 up to and with 49. The source set represents this for loop.

A node has a name and a number. The number is used internally for reference purposes. A node has access fields. These describe which elements of what array are used to store or read data from. This array is the array of the source program, and is used to construct data dependencies between nodes. These access fields in combination with the relations between nodes result in channels between the processes when converting the PDG to a PPN.

Each access field has a map property. For each element of the source set, the map specifies on what location of the array it must apply. The type property of an access field indicates if the

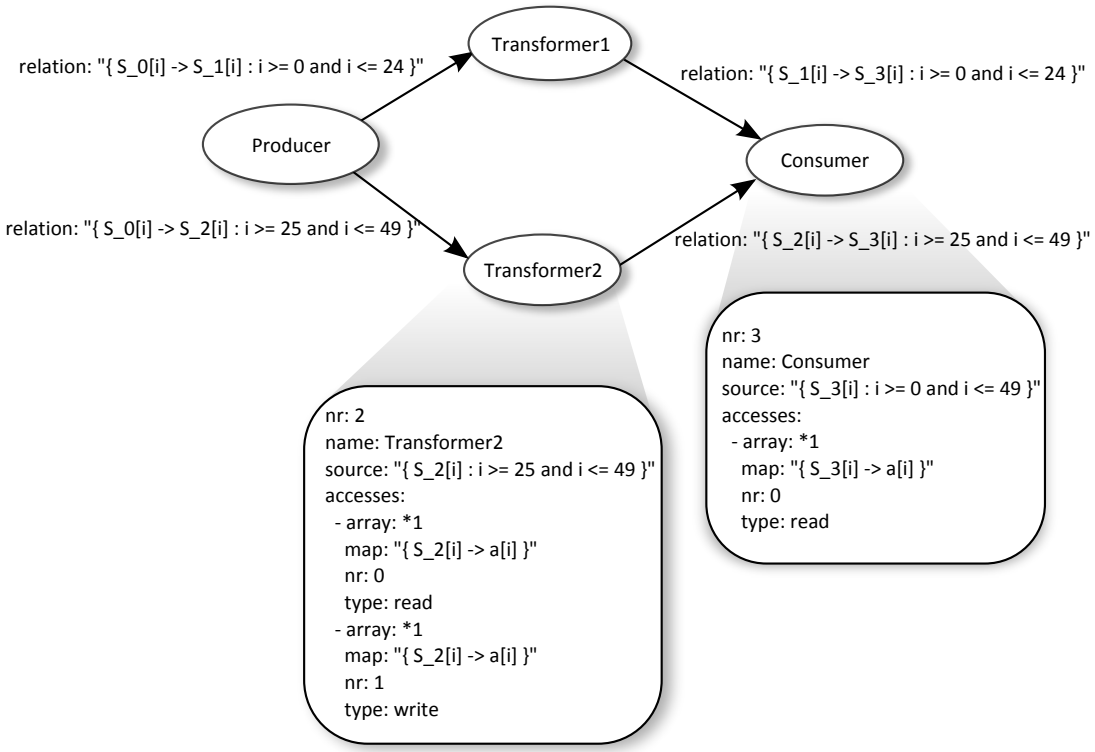

Figure 2.8: Detailed information available in a PDG.

access field is used for input or output. Some nodes only have input access, such as the Consumer node. If a node only has input access, it is a sink (the PPN equivalent is a process with only input arguments). A source is a node with only output access fields (or process with only output channels in a PPN).

2.5 Derivation of PDGs

A Polyhedral Dependence Graph (PDG) consists of a number of nodes connected by dependence relations.

The derivation of a PDG from C code (SANLP) goes in two steps. First, the program statements are translated into PDG processes (by the program `c2pdg`). The resulting PDG contains all nodes with their specifications but not yet the relations between the nodes. Note that the de-

pendencies can be implied from the nodes access fields. The PDG on this point however does not explicit contain dependency information. Despite this, they still are called polyhedral dependence graphs.

Second, the tool `pn` adds the dependency relations between the nodes (see Figure 2.1 for the tool flow). The resulting PDG is ready for the implementation phase. To implement a design in hardware, a designer can generate an FPGA project from the PDG. Alternatively, the designer can generate a simulation framework from the PDG, or a software project to run it on different types of processors. The implementation layer requires function latency information about the functions of the PDG. Some functions execute faster than others, and the difference impacts how fast a program can be executed. The function latency info indicates how much clock cycles each process requires to execute one iteration.

We demonstrate how the PDG is created during the tool flow with the C code of Figure 2.9 as an example.

```
int main() {
  int a[50];
  int i;

  for (i = 0; i < 50; i++) {
      Producer(&a[i]);
  }
  for (i = 0; i < 25; i++) {
      Transformer1(a[i], &a[i]);
  }
  for (i = 25; i < 50; i++) {
      Transformer2(a[i], &a[i]);
  }
  for (i = 0; i < 50; i++) {
      Consumer(a[i]);
  }

  return 0;
}
```

Figure 2.9: C code of a static affine loop program.

This program is a producer, transformer, consumer example with two transformers. The producer (Producer) creates data, the transformers (Transformer1, Transformer2) modify data and the consumer (Consumer) consumes the data. There is one one-dimensional array a declared that is used to transfer the data. In the function main the producer is called 50 times in the first for loop. The iterator of the for loop determines at which location of the array the data produced by the

Producer function is stored.

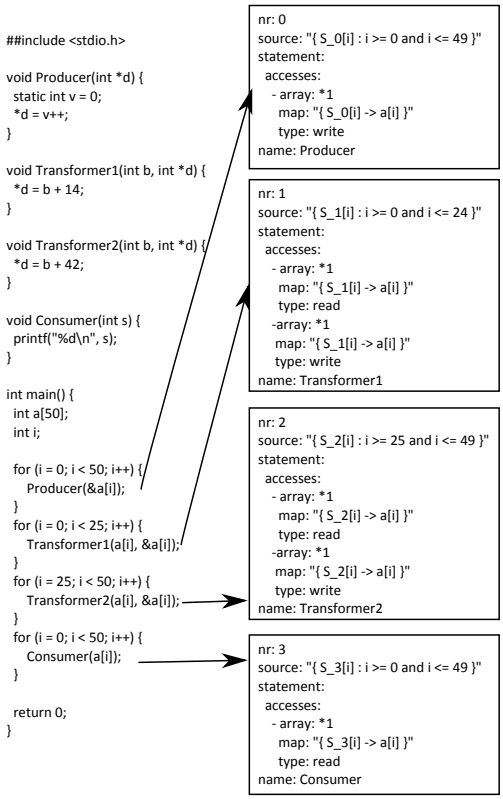

Figure 2.10: A program converted to PDG processes

When these statements are converted into a PDG, the Producer function will become a node. This node will have one access field, namely a write access to array a. It will have a source set describing the number of times this specific node will be executed. This is a translation from all for loops and if statements wrapped around the Producer function. In this case this is only one for loop. The resulting set here contains 50 one dimensional vectors, namely elements 0, . . . ,49. As discussed earlier, the Daedalus scheduler will make sure the set will be visited in lexicographical order, which is the order of executing the functions of the node. See also Figure 2.10.

When dependencies for a PDG are calculated using the `pn` tool, these are also available in the PDG. See the relation fields in Figure 2.11. Dependencies in the PDG describe how two nodes are connected. These dependencies contain a `from` and a `to` field, referring to the input and output of the dependence. Dependencies also indicate at what access point of the input and output node they

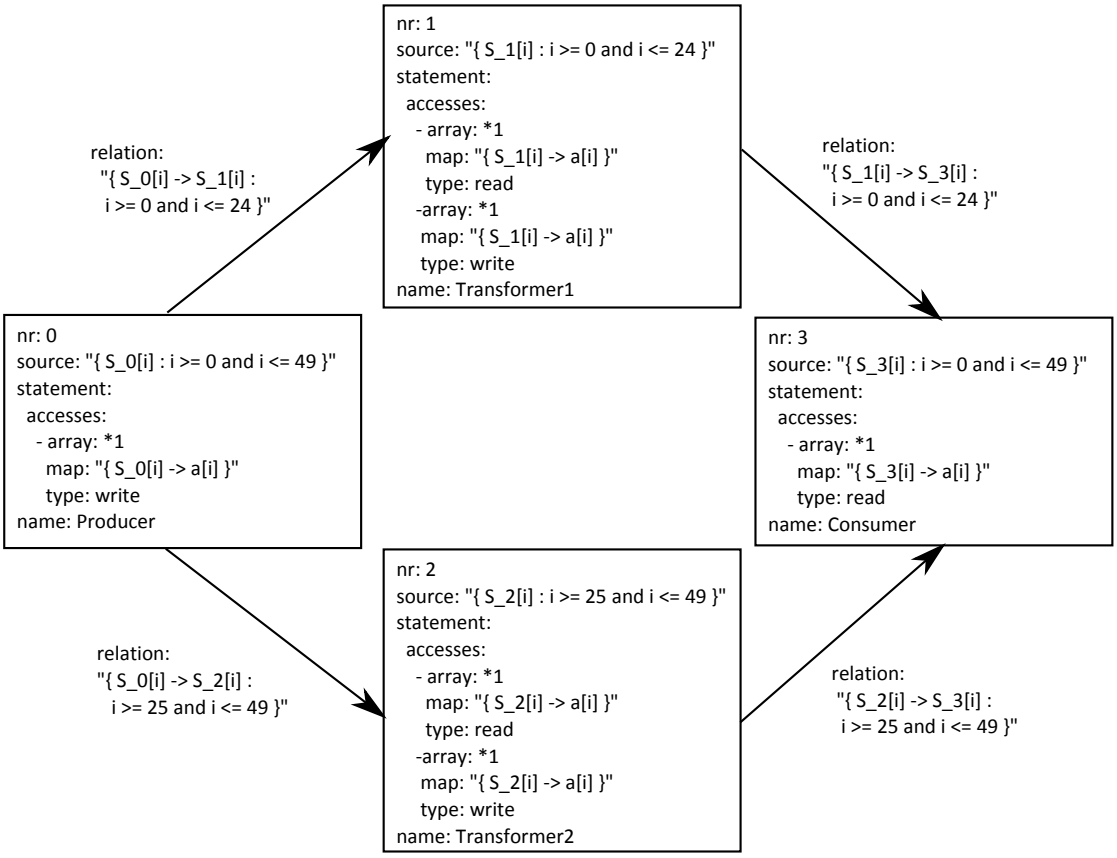

Figure 2.11: A program converted to PDG processes

are connected. The most important field of a dependency is the relation field. This field, visible in Figure 2.11 at the arrows, shows which set element of a node maps on which set element of another node. In the example the relation between the Producer node and the Transformer1 node is `S_0[i] -> S_1[i] : i >= 0 and i <= 24` . In this example all elements of the producer where $i >= 0$ and $i <= 24$ will be consumed by Transformer1 in the same order.

2.6 Data Dependencies

A process contains a set of elements representing its iteration points. Each iteration the process can use data from previous iterations and even from iterations of other processes. When an iteration uses data from a previous iteration, a dependency between them exists.

```
For i = 0; i < 4;i++
  For j = 0; j < 4;i++
    A[i,j] = F(A[i,j-1],A[i-1,j])
  End For
End For
```

Figure 2.12: Example of a SANLP with a two dimensional array.

In Figure 2.12 a two dimensional array A is used. For each iteration of the inner loop, a data location is assigned a new value. This value is determined by calling a function that uses two values of A. One of these values is assigned to A by the previous iteration of the inner loop. The other by the previous outer loop iteration. The assignment of A[i, j] is dependent on the values of A[$i, j-1$] and A[$i-1, j$].

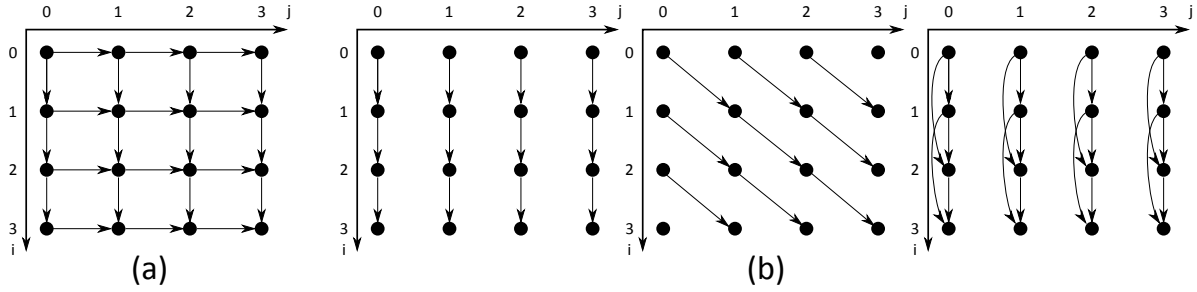

Figure 2.13: Iteration domains with their dependencies

Visualizing these dependencies we get Figure 2.13(a), the iteration domain of the process representing the code of Figure 2.12. Each iteration is indicated as a point. Arrows between the points indicate a data dependence between the two. The iterations that would require data from indexes out of bound, do not use this data. Every iteration point is a call to the statement `A[i,j] = F(A[i,j-1],A[i-1,j])` and the arrows show on which other statements it is dependent.

Dependencies can be between any two iteration points of a process, as demonstrated in Figure 2.13(b). Some of these iterations do not have any dependencies between each other. Then it is possible to execute both iterations in parallel.

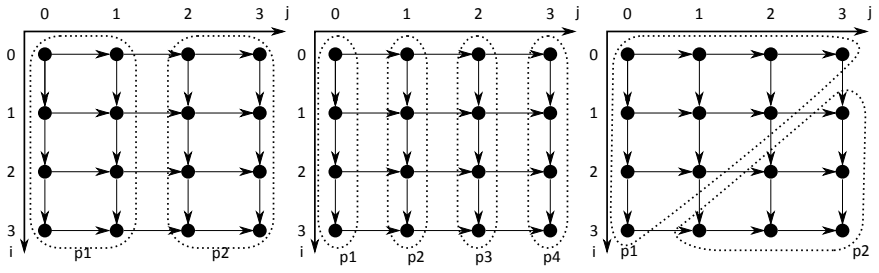

Figure 2.14: Various ways of dividing an iteration domain

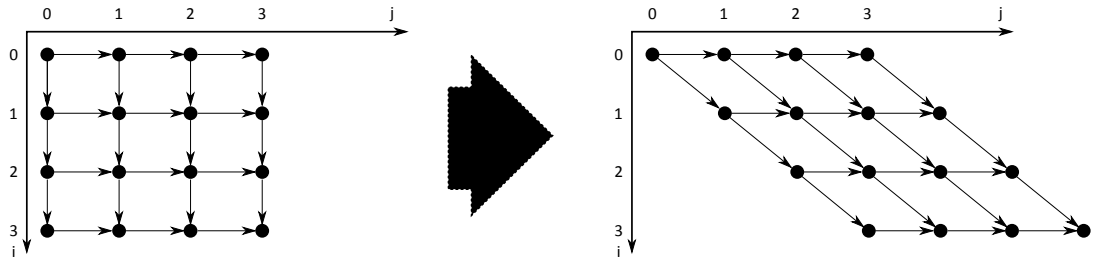

Figure 2.15: A skew transformation

2.7 Transformations and Partitions

A process executes its function a number of times. The executions and the exact data used for these executions are determined by the iteration points of the iteration domain of the process. The iteration domain can be transformed to achieve different behavior. For instance, using a skewing[10] transformation on the iteration domain of a process, the iteration points values are changed, such that iterations that could execute in parallel are easier to detect. See Figure 2.15 for the visualization of a skewing transformation on an arbitrary iteration domain.

In this thesis we will use transformations to split a process in multiple partitions. We will refer to the original process as the parent process of its partitions. A partition, which is a process, has the same properties of its parent process except that the partition's iteration domain is different. The iteration domain of a partition contains a number of iteration points that are also in the parent process. All the partitions of the same parent process combined contain exactly all iteration points of the parent process. Thus all function calls of the parent process are done by all partitions together.

Applying transformations on the iteration domain of a process, its iteration points can be

distributed between its partitions. In the next chapter, we will describe in detail how these transformations achieve this. In Figure 2.14, an iteration domain of a process is split by various transformations. The dots are iteration points and the arrows are the data dependencies between the iteration points. The dotted lines group the iterations points that are part of the same partition. The way the partition points are grouped influence the dependencies between the partitions. It is therefore important to note that different groupings of partition iteration points, i.e. different transformations on the parent process, are important for the data dependencies in the PPN.

Chapter 3

Transformations On PPNs

3.1 Tool Flow

A Polyhedral Process Network (PPN) consists of a number of processes, connected by channels. Implementing the PPN on hardware requires to map each process on a processor. Processes which are mapped on different processors can be executed in parallel. A process that takes a long time to execute may be split into smaller partitions. Splitting a process in different partitions makes the parallelism of the process explicit. Each of the partitions executes a subset of the function calls of the parent process, and only data dependencies between the partitions and other processes limit the execution. Each partition can be placed on a different processor, such that the partitions can execute in parallel resulting in a faster execution time of the program.

To split a process we split the corresponding node of the PDG in the tool flow. The illustration in Figure 3.1 shows where the `Transform` tool is present in the tool flow. First, C code is converted into a PDG. This PDG does not contain explicit dependencies between its nodes. While the dependencies could be implied from the access fields of the nodes, the dependencies are explicitly defined when the PDG is processed by the `pn` application.

The PDG without explicit dependencies can be transformed by the `Transform` tool. The new PDG changed by the `Transform` tool can continue the normal tool flow. The place of the `Transform` tool in the tool flow allows it to split nodes without taking care of its dependencies, which simplifies the implementation. These dependencies are not explicitly declared in the PDG on this point in the tool flow.

Nodes are converted into processes when a PDG is converted into a PPN. Splitting a node in the PDG results in a split of a process in the PPN. Splitting nodes in a PDG is relatively easy as almost all data fields can be copied to the splitted nodes. There are two differences. First, each new node needs a unique number. This can be arranged by taking the highest number used by all nodes in the PDG and increment it by one. Second, each node needs a new source set such that the combined source sets of the new nodes cover the whole source set of the original node without overlap. If there would be overlap, the new situation would run an original process iteration on different splitted processes (and thereby these processes would not be valid partitions of the original process). If not the whole source set of the node would be covered, some original process iterations would not be executed, since none of the new processes would execute this iteration. Splitting a process into partitions is accomplished by splitting the source set of its representing PDG node in a number of fully covered but not overlapping source sets. To create a partition, the parent process representing PDG node is copied, and its source set is replaced by one of the splitted source sets.

In this Chapter we will describe a number of methods to split a process. While the program applies the actual split on a PDG, the view of this chapter will be mainly on PPNs.

3.2 Methods

A process's iteration domain, conisting of iteration points, can be split into a number of subdomains. This can be done using different methods. First, one could say for each iteration point in which subdomain it should be placed. Second, an iteration domain can be split into two subdomains by a hyperplane. Every iteration point with a value below a given hyperplane goes in one subdomain, and every iteration point equal or above the hyperplane can be put in the other subdomain. This method can be extended to a list of hyperplanes. However, these hyperplanes should not intersect with each other, to prevent iteration points from occurring in multiple subdomains or not at all.

With non-intersecting hyperplanes and given a particular order of these hyperplanes, one can decide in what subdomain a specific iteration point must be placed. If the iteration point is smaller than the smallest hyperplane, it is placed in subdomain 1, if it is equal or bigger than the smallest hyperplane but smaller than the second smallest hyperplane, it is placed in subdomain 2. See Figure 3.2 for an example of a splitted iteration domain by three hyperplanes.

Figure 3.3 shows what can happen when planes intersect. Plane 1 intersects with plane 2. P1

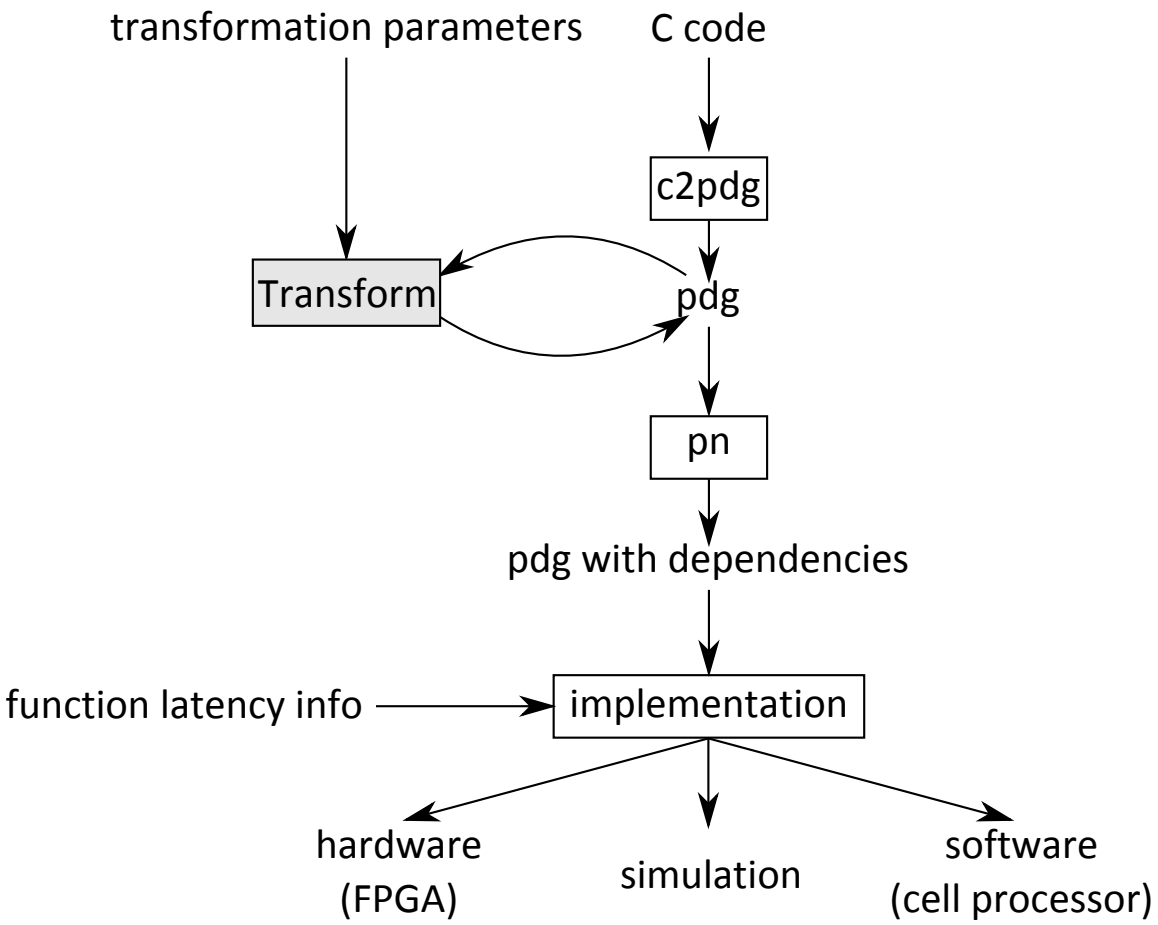

Figure 3.1: The tool flow for the transformation application

contains all iteration points smaller than plane 1. P2 contains all iteration points in between plane 1 and plane 2. P3 consists of all iteration points in between plane 2 and plane 3. Note that four elements are in P1 and P3. This is undesired, as splitting will be used to create partitions in a PPN, and partitions are not allowed to have overlap.

An iteration domain can also be divided by applying modulo equalities. in this case a modulo condition is applied on one dimension of the domain. All iteration points in the iteration domain which are a modulo on that dimension will be in one subdomain. Iteration points that have a remainder value of one for the same modulo expression go in the next subdomain. Figure 3.4 shows a domain which is split in two by modulo two on dimension j. All elements of the domain with

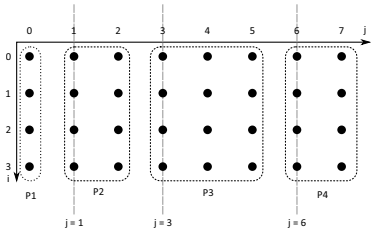

Figure 3.2: An iteration domain divided by hyperplanes

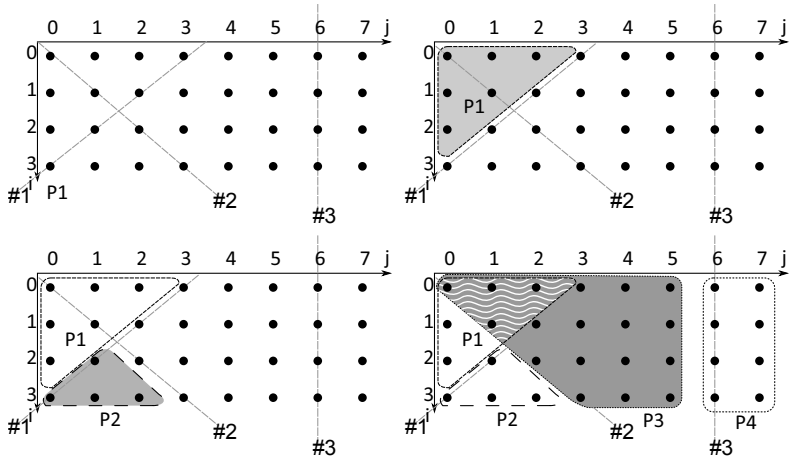

Figure 3.3: Splitting an iteration domain by the visualized planes, will result in a number of iteration points appearing in both P1 and P3

$j \ modulo \ 2 = 0$ are in subdomain P1, those with $j \ modulo \ 2 = 1$ are in subdomain P2. The definition for modulo splitting is as follows:

For an N-dimensional iteration domain a, the domain can be split based on a modulo x on dimension i
then $\{$
x new domains will be created,
Where elements in the new domains will contain all iteration points e of a where $e[i]$ modulo x = y
y is a different value $0, \ldots, x-1$ for all new domains.
$\}$

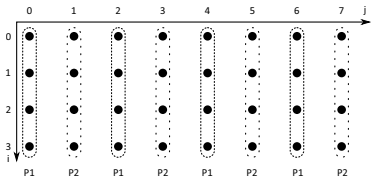

Figure 3.4: An two dimensional iteration domain split by the equality modulo 2 on j

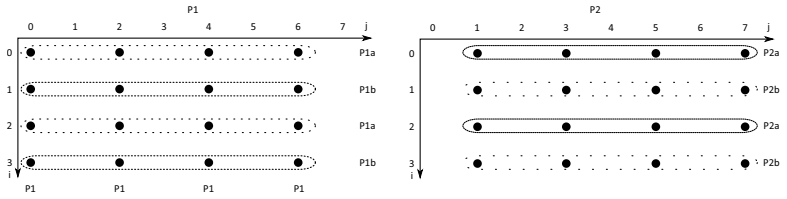

Figure 3.5: The two remaining iteration domains from Figure 3.4, with the modulo split applied on dimension i, splitting both iteration domains in two parts.

The modulo splitting method can be applied multiple times on different dimensions of an iteration domain. Note that after the first modulo is applied, x new domains are created. To apply modulo again on a different dimension, the new modulo split should be applied on all x domains. If the new modulo value is z, there will be $x \cdot z$ new domains after the second modulo is applied. In Figure 3.5, the splitted iteration domain of 3.4 is splitted a second time over $i \; modulo \; 2$.

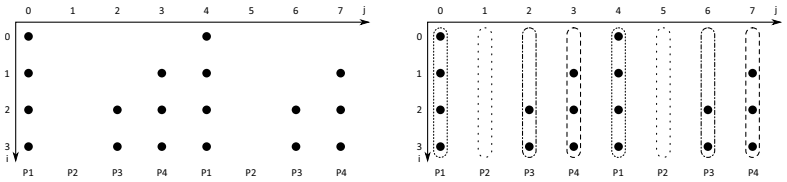

Figure 3.6: An iteration domain that is not a convex set, with the resulting domains if split by modulo split on dimension j in four parts.

If an iteration domain is a convex set, splitting it by a certain modulo x will create x domains. If an iteration domain is not convex, some of the created domains might contain more elements than others. Some domains might even end up completely empty. Consider the iteration domain from Figure 3.6. This iteration domain is split by $j \; modulo \; 4$, and the number of elements in each domain is distributed unevenly. Domain P1 has eight elements, while Domain P2 has no elements.

Process splitting has as goal to speed up the whole program. This can be done by giving every new partition its own processor to run on. If the source sets of the nodes representing partitions have an unequal distribution of elements, the computational load is not well distributed. Then, some processors may been terminated, while others have still a lot of work to do. It is therefore desired in most cases to split an iteration domain in partitions where the distribution of elements of their corresponding source sets is as equal as possible. As said before, this can be done by modulo splitting if the iteration domain is a convex set. In the case an iteration domain is not convex, or when one want to split an iteration domain differently, we can use plane splitting.

As already described at splitting based on hyperplanes, an iteration domain can be split by an ordered lists of hyperplanes. In which case the hyperplanes are given. For one dimension of an iteration domain, we can automatically generate this list. First, the iteration points in the iteration domain are counted. Knowing in how many pieces the iteration domain will be split in this dimension, an estimation of the number of elements that should be in the first new set can be calculated. Assuming the domain is evenly distributed, an approximation can be made were the split points should be. We have developed Algorithm 1 to search for the first desired split point. If the approximated split point creates a domain that contains to many elements, the split point value will be decreased. Alternatively, if it is too small it will increase the split point value. When an appropriate point is found, the new domain will be created. The algorithm continues with the second point, searching between the first split point and the end of the iteration domain.

Algorithm 1 describes how a domain is split based on split points of a single dimension. For a domain O that must be split in N parts, the algorithm searches $N - 1$ iterations for split points. The number of elements the next splitted domain should hold is calculated as described earlier and stored in te. Note that each iteration, the original domain is decreased by a newly found splitted domain. Thus the number of elements that the next new domain should hold is all elements from the decreased original domain divided by the number of parts left to split the domain in. A split point is found by a find function based on this adapted original domain. Using this split point we remove from the original domain the new domain just found.

The `FIND` function searches for an optimal split point tsp which splits the given domain O in two parts. One domain consisting of all elements with index $\leq$ tsp (the left domain), the other with all elements of index $> tsp$ (the right domain). The left domain should contain te elements. This is achieved by first calculating a guess where the split point potentially can be. In earlier versions of the algorithm the split point was guessed by taking the average of the lexicographical minimal and maximal point of the domain. Effectively, an evenly distributed domain would be

Algorithm 1 Split plane algorithm

Input: Domain O to be split in N parts
Output: An array with split points

1:
2: **for all** $1 \leq x < N$ **do**
3: te = $\lceil \frac{|O|}{(N-x+1)} \rceil$
4: $Sp[x]$ = FIND(te,O)
5: O = SPLIT(O,$Sp[x]$)
6: **end for**
7:
8: **function** FIND(te,O)
9: tsp = $\frac{te \cdot max(O)+(|O|-te) \cdot min(O)}{|O|}$
10:
11: L, R = SPLIT(O,tsp)
12: **if** $L == empty || R == empty$ **then**
13: **return** tsp
14: **end if**
15: **if** $|L| < te$ **then**
16: **return** FIND($te - |L|$, R)
17: **end if**
18: **if** $|L| > te$ **then**
19: **return** FIND(te, L)
20: **end if**
21: **return** tsp
22: **end function**

split in half by this guess split point. The current version of the algorithm calculates the split point based on where the split point should be if the domain was evenly distributed, i.e., if a domain should be split in three parts, the first attempt for finding the correct split point would be the value a third between the lowest and highest point values. When the possible location of tsp is calculated, O is split in a left (L) and right (R) domain as explained above. If either domain is empty, the split point is returned, since searching for a different split point is futile, this is the best split point. In case L contains an insufficient number of elements, the find algorithm is executed again, this time on the right domain, with as target elements te minus the already found elements in the left domain. These elements do no longer need to be taken into account because they already are included in the splitted domain, since they are located in the left domain. If the left domain is too big however, the find function is executed again with the same value as te on the left domain.

If L contains exactly te elements, the find function is done and returns tsp.

The algorithm can be applied on more than one dimension in the same way as the modulo algorithm can be applied multiple times. For a domain that should be split in x parts on dimension i and in y parts on dimension j we first split on dimension i the domain in x parts. We split each new domain on the jth dimension in y parts. Resulting in $x \cdot y$ parts. A disadvantage of splitting over each dimension individually is that it is possible that the final domains can differ in number of elements. Intermediate domains might have an equal number of elements, but getting as result that all created domains have an equal number of elements is not always possible because some elements might only be separated using an other dimension as split criteria.

Except splitting domains, it is also useful to be able to merge domains. For the examination of optimization we are going to do in Chapter 4, merging is not really important, as the focus is to increase performance by creating explicit parallelism. Merging different processes would be useful to save hardware, as it allows to place different processes on the same processor. Merging different processes is not straightforward to apply on the PDG, and therefore we do not implement it in the transformation tool.

We do implement the possibility to merge partitions which have the same parent process. Since all these partitions have the same properties except for their iteration domain, the merging process is simple. To merge partitions of the same process, their iteration domains are merged, the representing source set of this combined iteration domain is assigned to one of the partitions, as a result it becomes the reconstructed parent process. All other partitions are deleted, such that only the reconstructed parent process remains of all the partitions. This effectively undoes a previous transformation which created the partitions.

Chapter 4

Analysis Of PPNs

4.1 Introduction

4.1.1 General

In Chapter 3 a number of transformation methods have been discussed and implemented. These transformations can be used to increase explicit parallelism in a PDG. These explicit created parallelism can be used by the designer to explore different hardware configurations where processors are running only parts of the original processes, as the modifications on PDGs result in the corresponding transformations on PPNs. A designer can try some transformations, and see what the results are. A designer can also apply a transformation because the designer has some clues that a specific transformation will result in a speedup. Instead of letting a designer to find his own transformations that improve the application, we want to automatically apply efficient transformations. A method should examine the PPN and based on some properties of this PPN suggest a number of transformations that can lead to a better hardware implementation.

A better system is not the same as a faster system. In embedded systems it is not important that a system is as fast as possible, but rather fast enough. Any speed up once a system is fast enough is not necessary and might only be a waste of hardware resources. It is therefore important to be able to specify how much speed-up is desired for a program.

Finding results automatically saves the designer a lot of time. If the automation uses smart heuristics and algorithms, it is possible the automatic tool finds better solutions than the designer himself. Because an automatic process will eventually be faster, a larger part of the design space

can be explored to search for the best solution. After an automatic process has found a good solution candidate, a designer can use his domain knowledge to improve the solution.

We identify a number of problems for automatically detecting transformations that can be applied. Searching for the best solution the obvious question arises; what is the best transformation possible? This depends on the criteria used. For example, the question how much hardware is allowed to be used. Maximal speedup can be reached by giving each iteration point of each process a different processor. This configuration will be very costly, but guarantees the maximal number of iteration points to be executed concurrently. Except that it is hardware resource intensive, a lot of the increased resources are unnecessary as a lot of iteration points for each process can not run in parallel due to dependencies or could be run in sequence without a slower result. Another point of view could be to find a solution that would speed up the application just enough based on a given criteria, any hardware used more than required for the obligated speedup is in this case a waste.

As described in Chapter 3, splitting a process can be done in multiple ways, resulting in different distributed domains. In some cases one specific kind of split results in great improvements, while in another case it barely has effect. It is even possible that by certain splitting operations the solution gets worser. A method for suggesting splitting procedures should take into account the different effects of split methods.

4.1.2 Simplifying the Problem

Taking all criteria as described above into account results in a complicated problem. To simplify the problem one could try to find a method that can come up with a useful suggestion to transform a restricted set of PPNs such that the overall speed will be improved.

The simplified criteria are focused on speeding up a given program as fast as possible. Streaming applications often have sources with a fixed production rate, and sinks with a fixed consumption rate. These sources and sinks are in the PPN represented as processes that lack either input or output arguments. Since the rates of these processes depend on conditions outside the system, we do not split these source and sink processes. The purpose of this criterion is that the designer can use the rates of these source and sink processes to limit the maximal speed possible, effectively determining how much a program can be split up in multiple processes until it no longer has effect.

Hardware resources are the number of processors on the hardware implementation of a program. We assume that each process in the PPN will run on its own processor. Splitting a process

in multiple partitions, will increase the number of processors needed to implement the modified PPN, and thus the hardware resources cost. Communication delay between different processors or within the same processor can be different, therefore communication between different processes and within a single process can be different. Taking the different costs of communication into account adds extra complexity. In the simplified problem statement we assume the delays are included in the workload of the processes.

Increasing explicit parallelism could create a slower result in reality if the added communication between the new processors causes more delay than the parallelism improves the processing of data. This depends however strongly on the data communications that we assume are included in the workload for simplification. We therefore assume that increasing the parallelism in a PPN will not have a negative effect on the overall performance of the system.

The transformation methods of the previous chapter apply their transformations on integer sets representing iteration domains that are not parameterized. An integer set that is parameterized is expressed in equalities and inequalities containing parameters. With other words, it is not possible to express the number of elements these sets contain in integers without knowing the parametric values. Since we want to be able to analyze a known PPN configuration to improve by parallelism, as well as our `Transform` method can not handle parameterized sets, we will assume all sets representing iteration domains in an examined PPN do not contain are not parameterized.

One of the limitations for the criteria we use is to the take the number of processes created by split operations as a secondary objective. As described above, the main focus is to allow a program to execute as fast as possible, given the limitations of the source and sink nodes. If we do not care how many processes can be created, splitting each process for each of its iterations will result in the best solution. To prevent this, the simplified criteria imply that if more hardware resources do not improve the program total execution time, they should not be included. Using this criterion, the best solution is the one where splitting more processes is not improving the speed of the program, merging process partitions is slowing down the program and any other solution where this is the case is overall slower than this solution. Comparing a resulting solution with the maximal splitted up processes version allows evaluation if the maximal speedup is reached by the given solution. If the maximal split up program (excluding source and sink nodes) is faster, there exists a solution which is faster than the current solution.

Determining if processes can be removed is more difficult. A solution can consists of multiple processes split in different ways. Determining if processes can be removed would involve removing one process running a portion of an original process. The process can only be removed if the

iterations of the process are distributed among the other process partitions. In this thesis we focus on speedup by splitting processes and by doing this, distributing the pieces among new processes. To limit the number of processes one could try to merge different processes. However, the current tool flow makes this very hard, as merging occurs in the implementation layer, and not in the PDG. Merging is currently not the focus, but might be useful to consider in the future.

To determine if removing processes will not slow down the program, combining multiple different processes is not done. Only combining processes that originate from the same process are allowed, i.e., merging process partitions. There remains a large amount of possibilities on how a process could be merged to test for equal performance with less processes.

4.1.3 Tool Flow

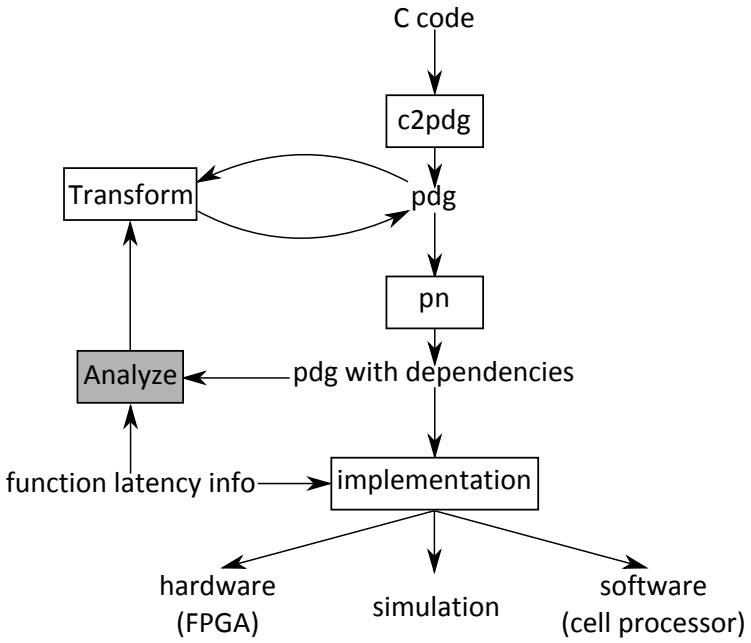

Figure 4.1: The location of the Transform tool in the whole modulation flow

The `Transform` tool uses PPNs without dependencies, that is, the associated PDG generated by `c2pdg`. On the other hand for the `Analyze` tool to find useful transformations, it is necessary that the dependence information in the PPN is present. Dependencies describe for each iteration of a process on which iterations of other processes (or itself) it is dependent. The `Analyze` tool

will return as result a number of transformation suggestions. These can then be applied by the `Transform` tool on the PPN without dependencies. Figure 4.1 represents the tool flow. C code is converted to a PPN file. this file can then optionally be transformed by the `Transform` tool. The `Transform` tool can either be use input from the `Analyze` tool or user input.

The PPN will receive its dependencies by the `pn` tool. Now the PPN can either continue to be implemented in a simulation, or used by the `Analyze` tool. In the `Analyze` tool the PPN with dependencies will be analyzed, returning transformation suggestions which can be used by the `Transformation` tool. The sequence `pn`, `Analyze`, `Transform` can be repeated multiple times. Each iteration the PPN will be split in more parts, based on the `Analyze` tool output. The `Analyze` tool can suggest a transformation, and analyze its effect the next iteration of the tool flow, and using the newly generated ppn with dependencies to come up with a new transformation.

4.1.4 Hardware Specifications

While a PPN gives a clear indication what the different processes in a program are, it lacks some information. Simulating the program requires that it is known how long a certain process takes for one iteration. This is expressed in clock cycles needed on a processor. This information is given by the designer in a different file, and used by the tools that make simulation possible.

4.2 Split Detection Method

The `Analyze` tool has as input a PPN with dependencies and a workload information file. The tool returns a number of suggested transformations, readable by the `Transform` tool. As described in the introduction of this Chapter, a transformation is an improvement if the resulting program executes faster. A transformation is unsuccessful if it either slows down the program or has no effect on speedup. If it has no effect, it is a bad transformation since the requirements state that the transformation suggestions should not waste hardware.

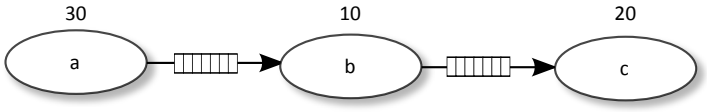

Figure 4.2: Example of dependencies between three processes.

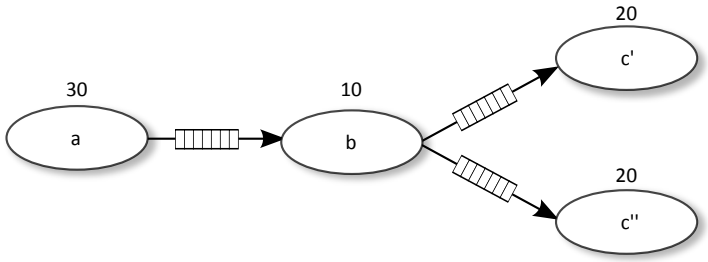

Figure 4.3: PPN of Figure 4.2 with process c split in c' and c''.

4.2.1 Determine Speed Up

For the `Analyze` tool it is important to know when transformation results in a speedup of the program. Splitting a process of the PPN will result in a speedup if its process partitions can run in parallel. To determine when partitions can run in parallel, the dependencies must be taken into account. If, for example, a process iteration is dependent on its previous iteration, splitting it will not result in a speedup, since the second partial process waits for the first partial process to finish and the third on the second etc. Dependencies are however not always this obvious. For instance an arbitrary process c can be dependent on process b and this process dependent on process a, see Figure 4.2 for a visualization of the example. The numbers above the processes show how much clock cycles a process needs to execute one iteration. Process b is a light process and can execute its iterations fast. Splitting c such that it can keep up with the iteration speed of b appears to be a good choice. In Figure 4.3 process c is split in c' and c''. If c' and c'' can run in parallel, each 20 clock cycles, two iterations can be processed. This effectively doubles the original speed of c, which was one iteration per 20 clock cycles. However, since a is a relatively slow process, b constantly has to wait for a. After b takes 10 clock cycles to execute one iteration, it has to wait for another 20 cycles before a is done with its iteration. As a consequence b can not execute faster than 30 clock cycles per iteration. The transformed c process (c' and c'') is now needlessly fast, and constantly waiting for b.

To determine which processes could be split to speedup the whole program, it is necessary to determine which processes slow down others. The PPN above was a simple example. Processes can be dependent on only a few specific iterations of other process instead of on every iteration. This changes the influence other processes have on a given process.

A file (as described in Section 4.1.4) contains the information on how fast each process per-

forms when not limited by its dependencies. Using these combined with the dependencies of the PDG, an analytical estimation can be made of the performance of each PPN process during a real program execution.

4.2.2 Throughput

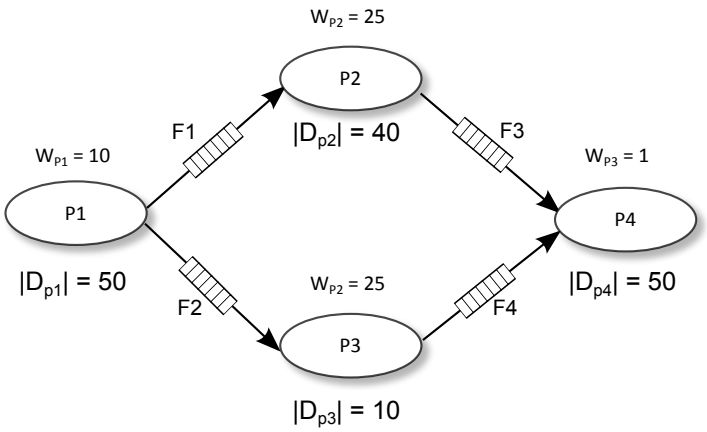

Figure 4.4: Example PPN

To detect how much a certain process can be accelerated, we must determine how the processes it depends on limit its speed. These processes can in turn also be limited by their dependent processes. A method for estimating these dependence limitations is the throughput analysis described in [11]. Throughput indicates how much data go through a FIFO channel, and how much data a process produces, for each clock cycle.

In a FIFO channel all data items stay in the order in which they were entered. Data that was first sent into the channel, will also first come out of the channel, hence first in first out. Since the throughput calculation requires channels to be FIFOs, we will assume when we discuss throughput, that channels are FIFOs.

Throughput calculations are done in topological ordering. Topological ordering in a PPN is the order in which processes are input dependent. A source process is a process without input arguments, and therefore is not input dependent and thus is first in the topological ordering. All nodes connected to the source are next in the topological ordering. The sink processes, which have no output arguments, are last in the topological ordering, as nothing is dependent on them, and are therefore not after a sink process. The problem with topological ordering is that if two processes

are dependent on each other i.e. one process receives input from the other, as well as send data to this process, it is more difficult to determine which of the processes is before the other. There is no topological ordering if there are processes dependent on each other, either directly, or by a path of processes, i.e., the PPN contains cycles. To calculate throughput, a PPN needs to have a topological ordering, we therefore take as criterion to use throughput that the used PPN does not contain cycles. We also do not allow processes in a PPN to contain self-loops, as this could also affect the behavior of a process not defined by the throughput.

A number of equations will be given to describe how to calculate throughput. It is important to keep in mind that the throughput of a process is based on its own processing speed and that of all its input dependent FIFOs. The throughput of a process does not include the limitations of its output arguments. These limitations will be taken into account by the FIFOs connected to the output arguments.

The throughput of a process P is the minimum of the aggregated incoming channel throughput T^{Faggr} and the isolated process throughput T_P^{iso}:

$$T_P = min(T^{Faggr}, T_P^{iso}). \tag{4.1}$$

T^{Faggr} is calculated based on the incoming channels to a process. For a single process with no input channels (a source process), T^{Faggr} is infinite such that the aggregation incoming channel throughput has no effect on the throughput of the process. The resulting calculation is $T_P = min(\infty, T_P^{iso})$. As noted earlier, the throughput of a process does not include its output argument limitations. Thus throughput is for a source process solely determined by its isolated throughput. The isolated throughput of a process is the amount of iterations per clock cycle:

$$T_P^{iso} = \frac{1}{W_P}, \tag{4.2}$$

where W_P is the workload of a process, the amount of clock cycles per iteration. The isolated throughput of process P1 of Figure 4.4 which executes one iteration in 10 clock cycles is $1/10$. Note that the isolated throughput is simplified. A more precise equation should take the read and write costs of the channels into account distinguishing between different and the same processes. However, as we stated in Section 4.1.2, with the simplified search criteria we assume that read and write time by processes is included in their workload, and do not need to be explicitly added in Equation 4.2.

The isolated throughput indicates how fast a process executes iterations if it is not blocked by

input or output channels.

The throughput of a channel f is the minimum of its read and write throughput.

$$T_f = min(T_f^{Wr}, T_f^{Rd}). \tag{4.3}$$

T_f indicates on average how much data per clock cycle goes through the channel. Since both read and write can block the channel, the slowest of the two indicates the total throughput. T_f^{Wr} can be calculated as follows:

$$T_f^{Wr} = \frac{|OP_P^j|}{|D_P|} \cdot T_P. \tag{4.4}$$

T_f^{Wr} is dependent on the process P that puts data in the FIFO channel. The write throughput is based on the throughput of this process, and how many of its iterations actually produce data for the FIFO. $|OP_P^j|$ is the amount of iterations of P that produces data for f. Note that j refers to the output argument of P to which f is connected.

$|D_P|$ is the domain size of P, which is the total number of iterations P will perform. T_P is the throughput of P.

In Figure 4.4, P1 has a domain size of 50. Only 40 of those iterations produce data for channel F1. $\frac{|OP_{P1}^j|}{|D_{P1}|} = \frac{40}{50} = \frac{4}{5}$. The throughput of P1 is $\frac{1}{10}$.
$T_{F1}^{Wr} = \frac{4}{5} \cdot T_{P1} = \frac{4}{5} \cdot \frac{1}{10} = \frac{4}{50}$.
To calculate T^{Rd} of the FIFO the following Equation is used.

$$T_f^{Rd} = \frac{|IP_P^j|}{|D_P|} \cdot T_P^{iso}, \tag{4.5}$$

Where $|IP_P^j|$ is the number of iterations a process consumes data from the FIFO channel. In Figure 4.4, the input size of P2 from F1 and the iteration domain of P2 are 40. Following Equation 4.2, the isolated throughput of P2 is $\frac{1}{W_{P2}} = \frac{1}{25}$. Using this information the read throughput of F1 can be calculated.
$T_{F1}^{Rd} = \frac{|IP_{P2}^j|}{|D_{P2}|} \cdot T_{P2}^{iso} = \frac{40}{40} \cdot \frac{1}{25} = \frac{2}{50}$.
The throughput of F1 can now be decided using Equation 4.3. $T_{F1} = min(T_{F1}^{Wr}, T_{F1}^{Rd}) = min(\frac{4}{50}, \frac{2}{50}) = \frac{2}{50}$.

Aggregation

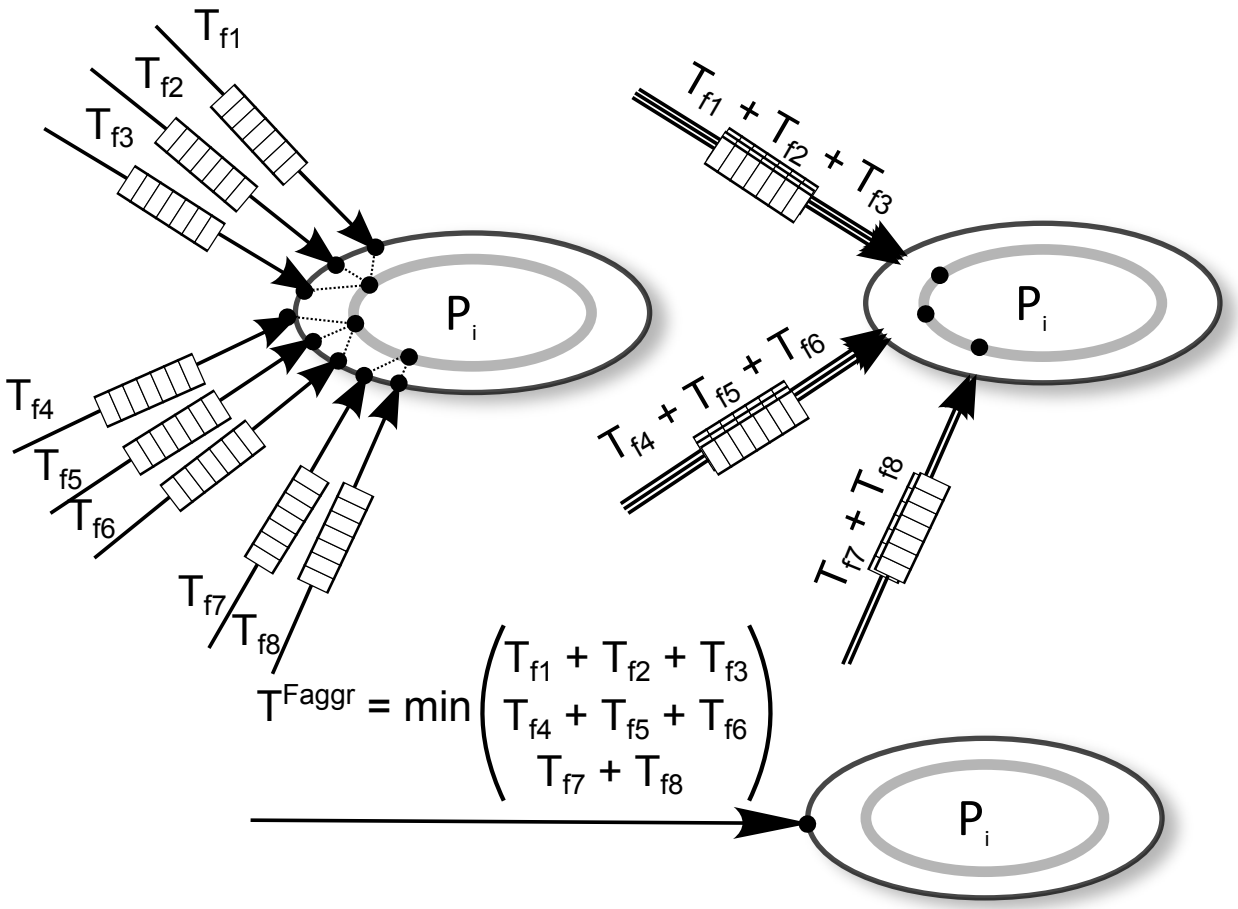

Figure 4.5: Example of how throughput aggregation is calculated

In Chapter 2 the concept of arguments of a PPN has been explained. To calculate the aggregated throughput of a process, all incoming FIFO channels that are connected on the same argument are taken together. Since they all contribute to the same argument of the process, their individual throughputs are added together. The sums of throughputs for all input arguments are compared. The lowest sum indicates the argument that limits the throughput of the process the most. The lowest sum is the aggregated throughput of the process. The aggregated throughput of a process is the rate of data it receives for one of its arguments, the argument which receives its

data the slowest of all the process arguments.

$$T^{Faggr} = min(\sum_{f \text{ connects to } j} T_f : \ j \text{ is an input argument of } P) \tag{4.6}$$

Figure 4.5 shows how aggregation is calculated for a process. All FIFO channels which are connected to the same argument of the process are added together. The minimum of these combined channels is the aggregated throughput.

4.2.3 Calculating Throughput

The throughput of a process is as Equation 4.1 states based on its aggregated throughput and its isolated throughput. The isolated throughput is calculated using the workload information of the process. The aggregated throughput is calculated by the throughput of the process input channels. These channel throughputs require the throughput of their input processes to be calculated. The processes that can be calculated without having to take input channels into account, are the processes without input channels. The source processes throughput can be calculated directly. To calculate all processes, the processes are evaluated in topological ordering based on their connections. If process a has a input from another process b via a channel, then b is before a in the topological ordering. Source processes are always first in the topological ordering and sink process are the last processes in the ordering.

Calculating the throughput of two processes with a loop between each other is impossible, since calculating the throughput of one of the processes depends on the throughput of the other and vice versa. We therefore only examine PPNs without cycles if we use the throughput calculation. This is a limitation, as many PPNs can contain cycles.

The `Analyze` tool uses an implementation of the throughput estimation algorithm as defined in Algorithm 1 of [11]. The algorithm calculates the throughput of all processes in topological ordering. For each process a number of calculations are executed. First, the isolated throughput is calculated, followed by the $T_{f_j}^{Rd}$ for all incoming FIFOs. Second, calculating all processes in topological ordering guarantees that all $T_{f_j}^{Wr}$ of all incoming FIFOs are calculated in previous steps (when calculating other processes' throughput). Third, the T_{f_j} for each incoming FIFO is calculated. Fourth, the algorithm calculates the aggregated throughput of the process using all incoming channel throughput values. Fifth, the throughput of the process is set as minimum of the isolated and aggregated throughput. Finally, the write throughput for all outgoing FIFOs is

calculated.

The final throughput value indicating how fast the program is, is the throughput of the last process in the topological ordering, i.e. the sink process.

4.2.4 Using Throughput

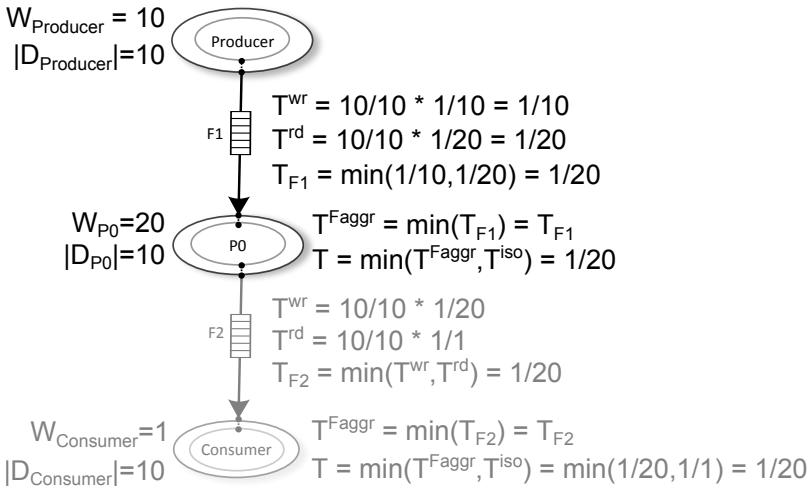

Figure 4.6: Throughput calculation of a PPN.

Throughput of a process is dependent on its aggregated throughput and its isolated throughput. Aggregated throughput is calculated using the connections of the input arguments. The output limitations of a process are not taken into account for its throughput. The output limitations affect the throughput of the channels connected to the output arguments of the process. For an individual process we only need to examine its input arguments, as the output limitations will be taken into account by successor processes.

A process should be split if the throughput indicates that the process consumes data slower than it receives data. In this case data accumulates in the channels connected to the input arguments. Once the channels are full, input processes will be blocked on writing. Splitting the process such that it executes in parallel increases the amount of data that can be processed each iteration.

Each channel has a write and read throughput as explained in Section 4.2.2. T^{Wr} indicates how many tokens each iteration enter the channel and T^{Rd} how much tokens leave the channel each iteration. If T^{Rd} is smaller than T^{Wr}, each iteration $T^{Wr} - T^{Rd}$ tokens are added to the

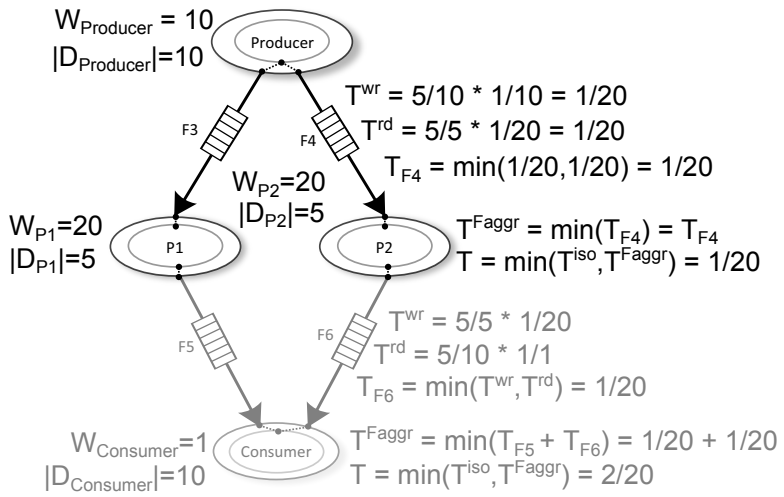

Figure 4.7: Throughput calculation and effect after splitting a process in two.

channel until it is full and blocks input processes. Splitting the process can prevent the blocking to occur, by allowing the data to be processed faster. By splitting a process in partitions not only the incoming channels are prevented from getting full, each process q receiving data from the partitions will receive its data faster. In the unmodified situation the parent process produced a number of data tokens for an input argument of q. In the modified situation, all partitions produce a portion of these data tokens for the same input argument of q. Thus q will receive its data tokens from different partitions that can produce this data in parallel and therefore q can have a higher throughput rate than when it would receive these tokens from the parent process. Assuming all processes can handle the improved throughput, the split decreases the execution time of the program. It is possible to split the process more than necessary. If a process is split such that for one of its input channels T^{Rd} is greater than T^{Wr} it has no improvement over the case where T^{Rd} is equal to T^{Wr}. When T^{Rd} is greater than T^{Wr} the process can process more tokens each iteration than it receives. At some point the channel will no longer contain tokens and the process has to wait until the channel has received new tokens. Splitting a process such that this situation occurs is a waste of resources, since it does not improve the throughput.

Example of Improvements based on Throughput

Using the throughput information, a process is selected to split. Splitting a process has to result in an improvement of throughput. Since throughput is based on the limitations of the process by its input channels, this is what will be examined when selecting processes.

If an examined process processes its input data slower than its predecessor processes produce this data, the examined process can be split to keep up with the production rate of the predecessor processes.

We show an example in Figure 4.6. The Producer process produces a data token every 10 clock cycles and process P0 consumes a token every 20 cycles. This results every 20 cycles in one data token processed by P0 and one data token added to F1 because P0 can not process it in the same rate as the Producer process could. Assuming the Consumer is fast enough to keep up with the P0 process, the P0 process is slowing the whole program down. Eventually the Producer will be output blocked as F1 gets full. On the right of Figure 4.6 the throughput calculations are displayed.

The workload of the Producer is 10, therefore the isolated throughput is $\frac{1}{10}$. Producer is a source process thus $T_{Producer} = T^{iso}_{Producer}$. All data tokens from Producer are sent to F1, $\frac{|OP^j_{Producer}|}{|D_{Producer}|} = \frac{10}{10}$. The write throughput of F1 is thus equal to the isolated throughput of Producer $T^{Wr}_{F1} = \frac{|OP^j_{Producer}|}{|D_{Producer}|} \cdot T^{iso}_{Producer} = \frac{10}{10} \cdot \frac{1}{10}$. Each iteration of P0, P0 consumes a data token from F1. All data P0 receives comes from F1. The workload of P0 is 20 and the isolated throughput $\frac{1}{20}$. The number of tokens read from F1 each iteration is $T^{Rd}_{F1} = \frac{|IP^j_{P0}|}{|D_{P0}|} \cdot T^{iso}_{P0} = \frac{10}{10} \cdot \frac{1}{20}$.

The write throughput of F1 is more than the read throughput, which is in correspondence with the described behavior; the Producer can produce tokens faster than P0 can consume. In Figure 4.7, P0 is split in processes P1 and P2. Both process one half of the domain P0 processes. The throughput of F4 is the same of F1. The write throughput of F4 is half of the write throughput of F1. It is halved because only half of the tokens the Producer creates are transported to P2 by F4. The other half of the tokens are transported by F3 to P1. Note that the throughput of P1 and of P2 is equal to the throughput of P0 of Figure 4.6. To detect if the split improves the PPN, we need to look at the throughput of the processes connected to the output arguments of the split process. The only output process of P0 is the Consumer process. In the bottom of Figure 4.6, the aggregated throughput of the Consumer is calculated. This is the throughput of F2, which is the minimum of its write and read throughput. The read throughput depends on the workload of

the Consumer, which is 1.The write throughput of F2 is calculated as the number of tokens of P0 designated for Consumer divided by the domain of P0 times the throughput of P0. This results in a write throughput of $\frac{1}{20}$ for F2. Comparing the throughput of the Consumer in Figure 4.6 with the throughput of the Consumer in Figure 4.7 will indicate if splitting P0 has effect on the PPN. The aggregated throughput here is the sum of the throughput of F5 and F6, as both send their tokens to the same argument of Consumer. The calculations of the throughput of F6 are shown in Figure 4.7. The write throughput is $\frac{5}{5} \cdot \frac{1}{20}$ since all 5 tokens P2 produces are intended for Consumer and the throughput of P2 is $\frac{1}{20}$. The calculations of the throughput of F3 is only different in that it applies on P1. The write throughput is smaller than the read throughput and determines the throughput of the FIFOs. In Figure 4.6, the aggregated throughput of Consumer results in $T^{Faggr} = min(T_{F_2}) = T_{F_2} = T^{wr}_{F2} = \frac{10}{10} \cdot \frac{1}{20}$. The throughput of the Consumer process is $T = min(T^{Faggr}, T_{iso}) = min(\frac{1}{20}, \frac{1}{1}) = \frac{1}{20}$.

In Figure 4.7, where P0 is split in two partitions, the aggregated throughput of the Consumer is $T^{Faggr} = min(T_{F_5} + T_{F_6}) = T_{F_5} + T_{F_6} = min(T^{wr}_{F5}, T^{rd}_{F5}) + min(T^{wr}_{F6}, T^{rd}_{F6}) = T^{wr}_{F5} + T^{wr}_{F6} = (\frac{5}{5} \cdot \frac{1}{20}) + (\frac{5}{5} \cdot \frac{1}{20}) = \frac{1}{10}$. The throughput of the Consumer process is $T = min(T^{Faggr}, T_{iso}) = min(\frac{1}{10}, \frac{1}{1}) = \frac{1}{10}$. Thus, the throughput of the Consumer process is doubled by splitting P0 in two parts.

A process split on the throughput not necessary has to result in an improvement as the process could still be limited by its output channels. These channels are however connected to other processes, and these could be optimized as well, with the exception of sink processes. Sink and source processes are as defined earlier artificial in that they represent the connection with the outer world. To influence the throughput calculations and thus the decisions taken by the `Analyze` tool, the designer could change the processing speed of the sources. These are taken into account since they are used for the throughput calculation, controlling what kind of transformations the `Analyze` will come up with.

Algorithm For Split Selection

Based on throughput calculations we present Algorithm 2 to determine which processes should be split, and in how many pieces.

Algorithm 2 examines every process in topological ordering, from last process till first process. Since throughput is calculated in topological ordering, splitting a process from last to first does not influence the throughput of processes earlier in the topological ordering.

Algorithm 2 Algorithm for splitting processes based on throughput.

Input: Throughput,T^{Faggr},T_{iso} for all processes
Input: T^{wr}, T^{rd} for all FIFOs
Input: P, set of all processes in the PPN.
Output: Specification which processes should be split.

1: **while** p = lastInTopo(P); P = P - p **do**
2: $\forall a \in p_{arguments} : cT_a^{Wr} = 0$
3: $\forall a \in p_{arguments} : cT_a^{Rd} = 0$
4: **if** $p == source || p == sink$ **then** ▷ Source and sink nodes are not allowed to be split
5: continue
6: **end if**
7:
8: **for all** $F \in p_{inputFIFOs}$ **do** ▷ Combine FIFOs T^{Wr} and T^{Rd} with same input argument
9: $cT^{Wr}_{F_{inputargument}} += T_F^{Wr}$
10: $cT^{Rd}_{F_{inputargument}} += T_F^{Rd}$
11: **end for**
12:
13: **if** $\exists a : cT_a^{rd} >= cT_a^{Wr}$ **then** ▷ slowest throughput caused by input, p should not be split
14: Continue
15: **end if**
16:
17: $speedupAmount = min(cT_a^{wr}/cT_a^{rd} \mid a \in p_{arguments})$
18: SPLIT(p,$speedupAmount$)
19: **end while**

For each process we first check if it is a source or sink. A source or sink process is skipped, as these processes are part of the connection with the outside world and their throughput is controlled by the designer to indicate how the outside world influences the program.

A set for combined write throughput (cT^{Wr}) is created and one for the combined read throughput (cT^{Rd}). These sets contain for each input argument of the process the sum of all T^{Wr} and T^{Rd} respectively, of all FIFOs connected to that argument. This is done on code lines 8 till 11 of algorithm 2.

For an arbitrary process a with FIFOs F_1 and F_2 connected to input argument 1 of a, $cT_1^{Wr} = T_{F_1}^{Wr} + T_{F_2}^{Wr}$ and $cT_1^{Rd} = T_{F_1}^{Rd} + T_{F_2}^{Rd}$.

If for a process there is an argument a where $cT_a^{rd} \geq cT_a^{Wr}$, the total data designated for that argument is faster consumed than produced. Improving the consumption rate by splitting the

process will not improve the throughput, as the input will stay the limiting factor. The algorithm will therefore not split processes where this is the case. All processes such that for all arguments the cT^{Wr} is bigger than cT^{Rd} can be split to comply with the combined write throughput. The process should get $min(cT_a^{wr}/cT_a^{rd} \mid a \in p_{arguments})$ times faster, resulting in processes where at least one argument gets limited by the write throughput instead of the read throughput. The SPLIT function will be explained in Section 4.2.5. This function splits a process p in $min(cT_a^{wr}/cT_a^{rd} \mid a \in p_{arguments})$ parts. We assume that the process partitions can run in parallel, and give exactly the amount of speedup equal to the number of times the process is split. Since self dependencies and loops are not supported by our current algorithm, the only limit on the concurrent behavior is the sequential production of data designated for the original process.

4.2.5 Splitting

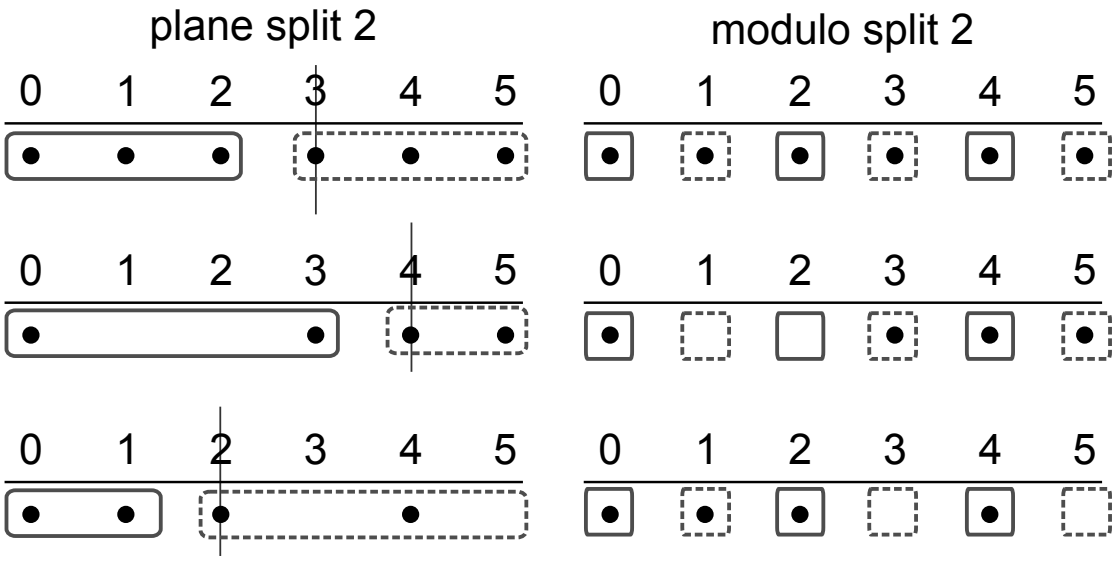

Figure 4.8: Example of iteration sets split by the Transform tool.

In Section 4.2.4 an algorithm is presented that indicates which processes should be split, and how much faster the partitions together should process the original data. As described in Chapter 3 a process could be split in multiple ways. The `Analyze` tool does not actually split processes, but generates a command line that contains instructions for the `Transform` tool how the PPN should be split.

We consider two options for splitting a process. We consider plane splitting and modulo splitting. In both cases the `Transform` tool can split a process in a desired number of parts. In the case of plane splitting using Algorithm 1 the number of iterations in each partition will be distributed as equal as the algorithm is able to do, taking non convex iteration domains into account.

The modulo split transformation will just split on a given dimension using modulo conditions. This results in a problem. Modulo split does not modify its split behavior based on the distribution of the iteration points in an iteration domain. This can result in partitions with large differences in their iteration domains. If for instance all iteration points have the same modulo as the transformation tries to split in, one partition will contain all iterations, while the others will be empty. In Figure 4.8, three different sets are split by both methods. In the case of plane splitting, the algorithm behind it will set the split point such that the partitions are equal in size. The modulo split will split on the given modulo, here 2. This results in all iterations with value modulo two equals zero are in one set, and all values modulo two equals one in the other set. In the figure it is clearly show that this can result in unevenly distributed partitions. Even if these partitions run in parallel, their uneven size does not give the desired result of increasing the processing speed by the amount of times the process is split in.

The plane split method is also not perfect for the purpose of speeding up a process consumption rate. In the top left example of Figure 4.8 one partition contains iteration points 0,1 and 3, where the other contains iteration points 3,4 and 5. The first three iteration points (in lexicographical order) of the original iteration domain are all in the first partition, and the next three iterations in the second partition. In the modulo split example of Figure 4.8 (the upper right one), looking at the iterations in lexicographical order, the iterations alternate between the two partitions. The first iteration point (0) is in the first partition, the second iteration point (1) is in the second partition. The iteration points of a process are executed in lexicographical order once implemented by the Daedalus scheduler. We assume processes are connected with FIFOs to each other. Data comes out of a FIFO in the same order as it comes in. A process consumes its data in the same order it is written to the FIFO. If a process is split by a plane split transformation, the first x data tokens designated for the parent process will go to the first partition, the next y data tokens will go to the next partition etc. This can prevent the partitions to run in parallel, as explained in the following example.

In Figure 4.9(a), a producer process generates data tokens designated for process A. It starts with the creation of t[1], a data token that is required by the first iteration of process A. It sequen-

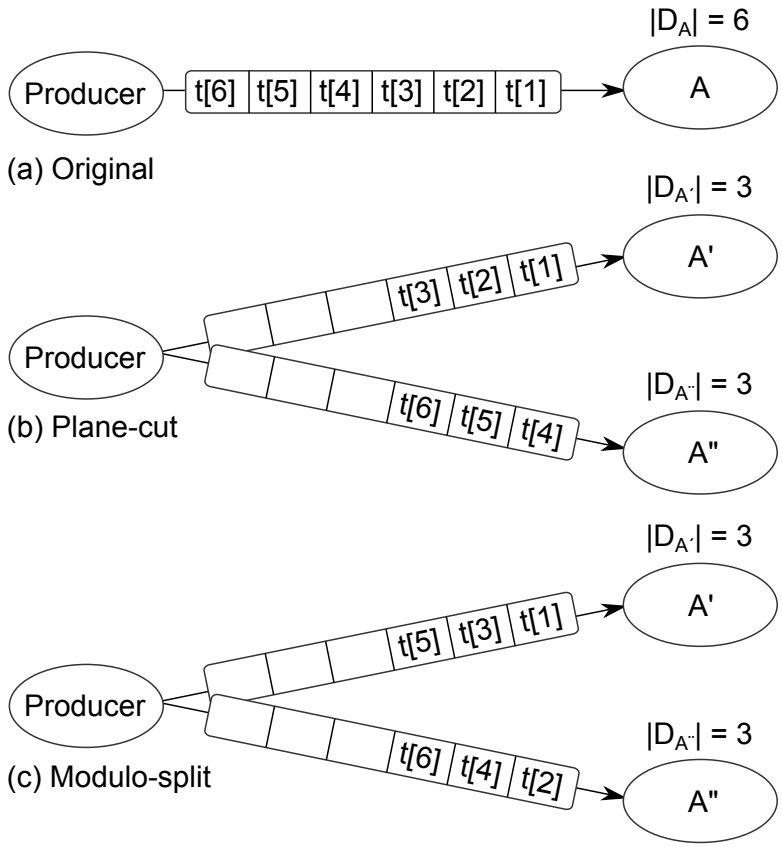

Figure 4.9: Order of data tokens produced by a process

tially continues with t[2], t[3], etc, designated for iteration 2, 3 of process A respectively. In Figure 4.9(b), process A is split in A' and A'' with the plane split method. A' processes iterations 1,2, and 3 of the original process A and A'' processes iterations 4,5 and 6 of A. Data token t[4] is created by Producer after it has created tokens t[1],t[2] and t[3]. A'' has to wait until all tokens for A' are created before it starts receiving the tokens it needs to execute. A' and A'' can only start to run in parallel when the producer has created all the tokens for A'. If the FIFO between the producer and A' can store less tokens than the producer needs to produce before starting its tokens creation for A'', the producer might get output blocked by A' and has to wait for A' before it can start producing tokens for A''.

In Figure 4.9(c), A is split by the modulo split method. Here A' contains all iterations where the values modulo 2 equal 1, and A'' where the values modulo 2 equals 0. Now after the producer

has created one token for A', the next token is designated for A". A" can thus start executing after the producer has produced one token for both split processes instead of all tokens for A'.

In the implementation layer of the tool flow the sizes of all FIFOs are calculated, such that no deadlock occurs. The FIFO sizes are just large enough to store sufficient data to let all processes at some point execute. This means in the case of 4.9(b) the FIFO sizes are one, as only one token is required by A' each time (the same counts for A"). Blocking of the production process would then occur after it has produced t[1]. Once A' starts consuming t[1], then t[2] can be created. This greatly limits the overlap where both A' and A" receive tokens (to execute in parallel) greatly. The preferred split method is therefore the modulo split and not the plane split.

To allow the `Analyze` tool to apply the modulo split method without ending up in highly uneven iteration points distributed sets, we assume all the sets representing the iteration domains of the PPN the `Analyze` tool examines are convex sets. As stated in Section 3.2 the modulo split method works well on convex sets, and streaming applications have only rarely iteration domain representing sets that are not convex sets.

4.2.6 Merging

The modulo split is limited by processes with iteration sets that are not convex sets. For the kind of programs the tool flow is designed, these iteration sets are not common, and we therefore do not attempt to address this problem. The `Analyze`, `Transform` and `pn` tools can run in sequence repeatedly. This is encouraged as with each iteration based on the new throughput values of split processes earlier in the topological ordering, the `Analyze` tool can better approximate how many times a process later in the topological ordering can be split. The bottleneck process may change after splitting. Therefore, an iterative approach is needed such that the `Analyze` tool can take the previously suggested transformations into account.

At each iteration of the tool flow processes can be split up by the modulo split method. After the first iteration we no longer can assume that all process iteration domains are convex sets. See Figure 4.10, after the set is split in a set with the equation modulo two equals zero and a set with the equation modulo two equals one, both created sets are no longer convex.

In order to apply a modulo on an already split set, we merge all partitions together and then split this combined process by the new modulo value. This not only allow modulo to split on convex sets it also allows a previous split process to change the modulo splitting factor, from 2 into 3 for instance. Would in this case the partitions not be merged, only all partitions could be

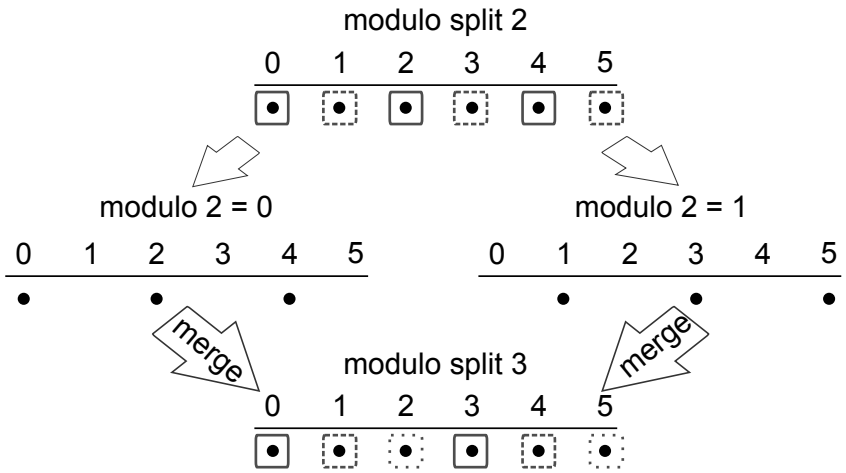

Figure 4.10: Example of a process iteration set that get split twice by the repeated tool flow of Transform pn Analyse.

split separately in more pieces, instead of the original process in equal parts. If the partitions are not merged in Figure 4.10, it is not possible with only modulo split applied on the individual partitions to get three even sized partitions.

Note that merging is in this special case supported by the `Transform` tool, as merging of partitions is supported contrary to merging different processes that do not stem from a common process.

Chapter 5

Experiments

5.1 Introduction

In this chapter we demonstrate what effect the implemented `Transform` and `Analyze` tools have on a program. The PDGs of the programs are modified by the `Transform` tool. Then simulated with the implementation tools available in the Daedalus tool flow. The `Analyze` tool generate commands for the `Transform` tool to execute, aiming for a higher throughput.

5.2 Splitting Processes

5.2.1 Sobel

Sobel[13] is an algorithm which uses a convolution matrix to detect edges in an image. In Figure 5.1, we show the PPN of sobel that consists of five processes. readPixel is the only source node, and represents the reading of pixels of an image from disk or camera stream. The two gradient processes apply a matrix manipulation on some of these image pixels. One gradient process applies on pixels that are horizontal next to each other, the other gradient vertical. The gradient process require a number of tokens of readPixel for its first iteration. While waiting for these tokens, additional tokens from readPixel arrive at the FIFOs of the gradient process required for later iterations. The FIFOs between readPixel and gradient need to have sufficient storage capacity to store these tokens. The absVal process combines the results from the two gradient processes. The writePixel is the sink process, and represents the storing of the pixels in a new image or writing to

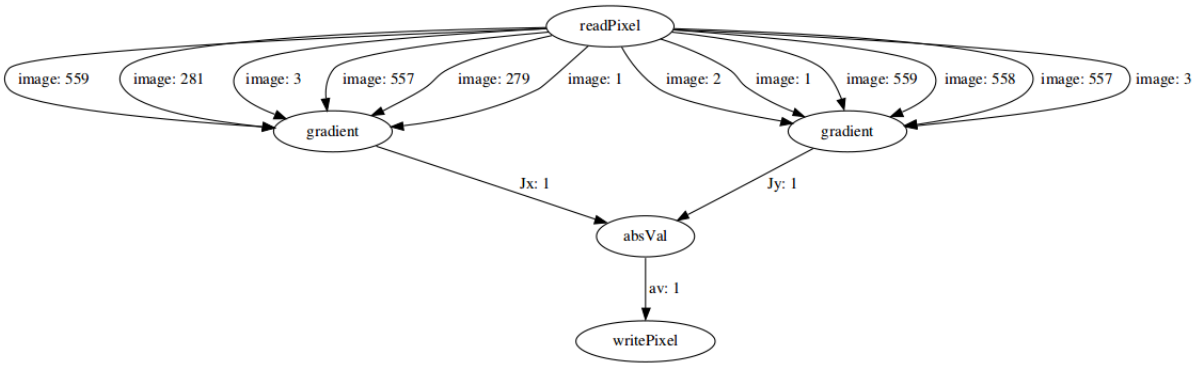

Figure 5.1: The PPN of the sobel program

a display device.

The workload for the different processes are the same as those used in [13]. The read and write Pixel processes have a workload of 5, the gradient processes have a workload of 31 cycles for each iteration and the absVal process has a workload of 118 cycles. The latencies for writing and reading from FIFOs are 1, which we assume are included in the workload.

In this section we apply the transformation tool manually, that is, we select the transformation parameters manually. The graph of Figure 5.2 shows how fast the Sobel program is after the absVal process is split by the transformation tool. With number of clock cycles on the y axis and number of partitions on the x axis.

The processes in the sobel program have a two dimensional iteration domain. 'plane dim 1' in the graph are the data points where the absVal process is split by a plane on the first dimension. The graph shows that this splitting method, even after splitting it in 6 pieces, has almost no effect. The split program executes in 99.98% of the time the original program executes. 'plane dim 2', which is plane split over the second dimension, has more effect. The program split by this method executes in 96.39% of the original time.

This improvement of speed occurs when split processes can execute in parallel. For 'plane dim 1' the first dimension corresponds with the outer loop of the original program. Splitting this outer loop, each partition runs a part of the outer loop, with the corresponding inner loop. Overlap is present when one partition is busy with the last iteration of the outer loop and the last iterations of the corresponding inner loop and the next partition starts its first outer loop and the corresponding

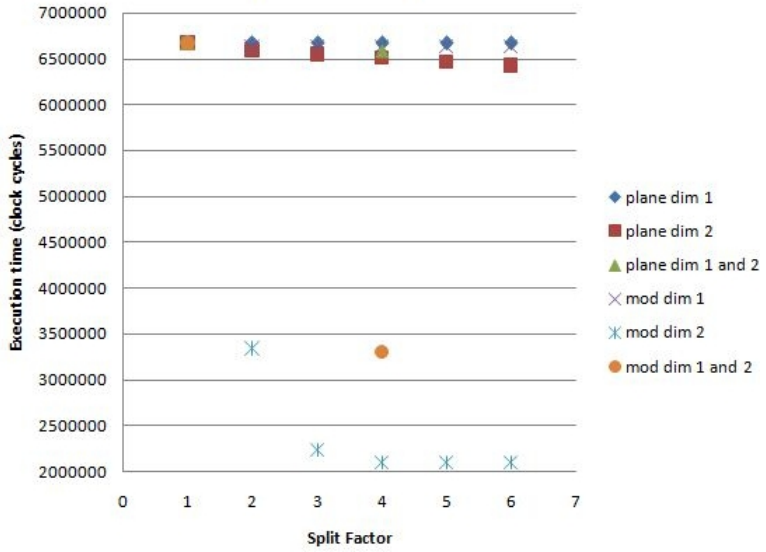

Figure 5.2: Number of clock cycles needed for Sobel to complete after the absVal process is split.

first inner loop iteration. The size of the actual overlap is based on the buffer size of the FIFO to the first partition such that the producing process (readPixel) can start producing for the second partition. Since FIFO sizes are limited, this is 200 clock cycles for each overlap. And each split on the outer loop results in one additional overlap where two partitions run in parallel.

'plane dim 2' on the other hand, is split on its inner loop. Every outer loop iteration, all partitions are used. First, the partition with the first part of the inner loop, continued by the partition with the second part etc. In 'plane dim 1' the number of overlaps is equal to the amount of partitions minus 1, in 'plane dim 2' the overlap amount is equal to the number of partitions times the size of the outer loop minus one. Therefore, 'plane dim 2' is more effective.

The difference between 'mod dim 1' (process absVal split by modulo split on dimension one) and 'mod dim 2' (process absVal split by modulo split on the second dimension) can be explained in the same way.

'mod dim 2' is a lot more effective than 'mod dim 2', although both split methods apply on the inner loop. In the case 'mod dim 1' is split in three pieces, each partition takes a third of the iterations of the inner loop to process. The amount of overlap in the inner loop is thus the number of partitions if it is not the last iteration of the outer loop else number of partitions minus one. In case of the modulo split on the inner loop in x pieces, every sequence of x iterations is processed

by different partitions. Effectively this means more overlap. Thus x data tokens designated for the absVal process can be processed in parallel. There is delay between each data token based on the input, and if the $x + 1$ data token arrives before the first token is processed, it can not be processed immediately.

'plane dim 1 and 2' shows results where absVal is split both in dimension 1 and dimension 2, where the total number of partitions is on the x axis. Splitting the process twice in the fist dimension and twice in the second results in four partitions. 'mod dim 1 and 2' shows the results where the process is split by modulo split in both dimension 1 and dimension 2. Both are in between the same method full split in dimension 1 and full split in dimension 2.

'mod dim 2' saturates when the split factor is more than 4. This can indicate that another process becomes the bottleneck for the program at this point.

We also try to improve the program by splitting the gradient nodes. Splitting one of them has no effect, nor has splitting both. This is expected: as splitting absVal make the program execute in only 31.47% of the original number of cycles, the absVal process is most likely the bottleneck.

Splitting the absVal in four with the modulo split method on dimension 2 prevents absVal from being the bottleneck. Splitting one of the gradients after this has at most an improvement of only 5 clock cycles. The absVal process requires input from both gradient processes. If one of them became faster than the other, absVal still must wait for input from the slower of the two.

In Figure 5.3, the absVal process is first split by modulo split in four partitions on dimension 2. Then both gradients are split by either plane or modulo split in either dimension 1 or 2 (or both).

The results are similar as that of Figure 5.2. In both graphs 'plane split dim 2' has effect, and 'modulo split dim 2' has a huge effect. 'modulo split dim 2' stops improving after 3 splits. Probably because another process (absVal) is the bottle neck of the program.

Splitting sobel in five absVal partitions on dimension 2 with the modulo method, and both gradient processes in four partitions (also with the modulo split on dimension 2), the program execution time decreases again, to 1347889 clock cycles. Thus the gradient nodes were indeed limited by the absVal node.

5.2.2 Correctness of Splitting

Splitting and improving a program is a good thing to do as long as the program stays functioning correctly, i.e. satisfying the original program specification. In the case of sobel the output is a picture. If a transformation is applied on the sobel program, the program should still output the

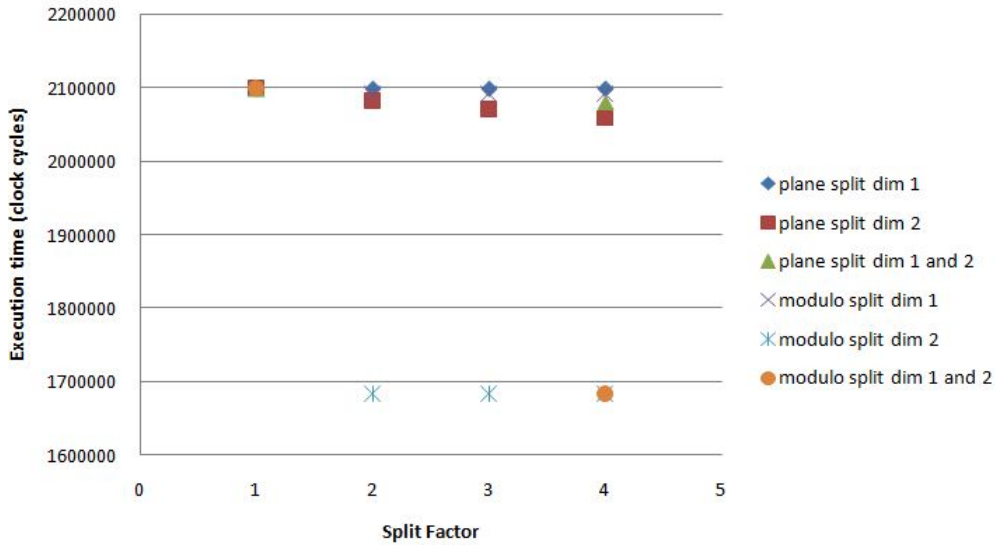

Figure 5.3: Results for splitting the gradient nodes (after splitting absVal with modulo split in 4 pieces on dimension 2.)

exact same picture. We tested various transformations, and compared the output data. In all cases the output picture was exactly the same as the output picture of the unmodified program.

5.2.3 QR Decomposition

The QR decomposition algorithm can be used to solve systems of linear equations[14]. The program consists of two source processes. In Figure 5.4, we show a visual representation. A zero process and a read_x process provide data for the program. The vectorize and rotate processes receive input from the sources and each other. The vectorize process has a two dimensional iteration domain, and has a self loop. The rotate process consists of a three dimensional iteration domain and has three self loops. The vectorize and rotate processes write data designated for each other, themselves and the sink process. The sink process only consumes data.

In Figure 5.5, we show the execution time of the QR program for various split commands. On the x axis we show on what dimension the split command applies. Zero on the x axis indicates no split. On the right side we show the different methods that are used. A PPN can be split in many ways, and many times. In this example we compare the effect of splitting over different dimensions. The vectorize process is split by plane and modulo on dimension 1 and 2. Since the

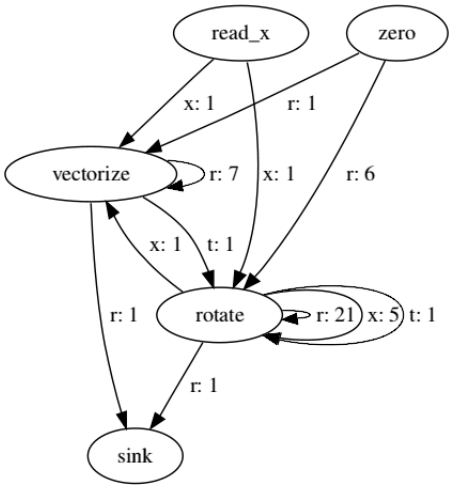

Figure 5.4: PPN of the QR algorithm

rotate process has three dimensions, this process is split by plane and modulo on dimensions 1, 2 and 3. All splits are a factor 2.

'Modulo vectorize' are the results where the vectorize process is split in two by modulo split. Contrary to the results of the sobel, splitting on the outer loop iteration domain has more effect than splitting on the inner loop. Splitting on dimension 2 has no effect at all.

'Modulo rotate' are the results where the rotate process is split in two by modulo split. Here splitting on dimension 1 has no effect. Splitting on dimension 3 results in an execution time of 2185 clock cycles less than splitting on dimension 2.

'Modulo vectorize & rotate' are the results where the vectorize and rotate process are split by the modulo split method on the dimension indicated by the x axis. Splitting the vectorize and rotate process on dimension 1 yields the lowest execution time. This indicates that splitting only the rotate process on dimension 1 has no effect because it gets blocked by the vectorize process. Splitting the vectorize and rotate process on dimension 2 results in a program only one clock cycle faster than when only the rotate process is split over dimension two with the modulo split.

'Plane vectorize' are the results where the vectorize process is split with a factor two by plane splitting on the dimension indicated by the x axis. In this case plane splitting on dimension 2 has effect, where modulo split on dimension 2 has none.

'plane rotate' are the results where the rotate process is split by the plane split method with factor two on the dimension indicated by the x axis.

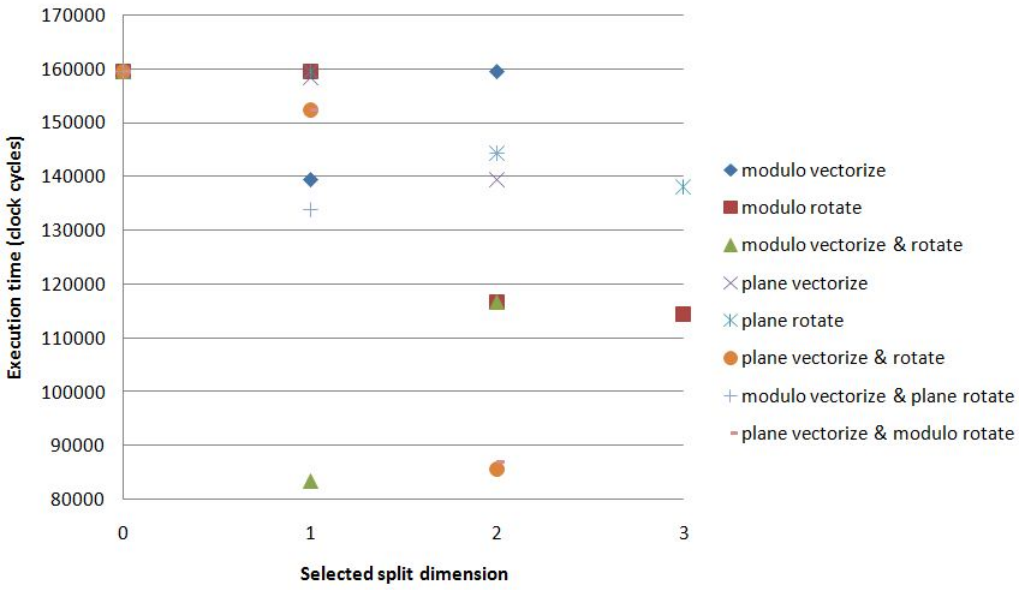

Figure 5.5: Effect of different split parameters on QR

'plane vectorize & rotate' are the results where the vectorize and rotate process are split by the plane split method with a factor two on the dimension indicated by the x axis. Splitting on dimension 1 decreases the program execution time by 7088 clock cycles, splitting on dimension two by 73707 clock cycles.

'modulo vectorize & plane rotate' are the results where the vectorize process is split with a factor two by the modulo split method and the rotate process is split with a factor two by the plane split method.

'plane vectorize & modulo rotate' are the results where the rotate process is split with a factor two by the plane split method and the rotate process is split with a factor two by the plane split method. When split on dimension 1 the results are equal to 'plane vectorize & rotate'. When split on dimension 2, the resulting program executes 1172 clock cycles longer than the 'plane vectorize & rotate' variant.

These results show that not in all cases splitting on the most inner loop gives the best results. The QR program split on dimension 1 with 'modulo vectorize & rotate' yields lowest execution time of our results, with 83506 clock cycles. The split method also affects on which dimension the split can best be applied, comparing 'plane vectorize' with 'modulo vectorize', for plane split

dimension 2 is better, where for modulo dimension 1 is better.

5.3 Analyzing Programs

In this section we use the `Analyze` tool as input for the `Transform` tool. The `Analyze` ,`Transform`, `pn` sequence can be run multiple times to improve the final result. Since plane splitting performs worse than modulo split on average, as explained in Section 4.2.5, the `Analyze` tool generates modulo split commands and not plane split commands.

5.3.1 Producer Transformer Consumer Program

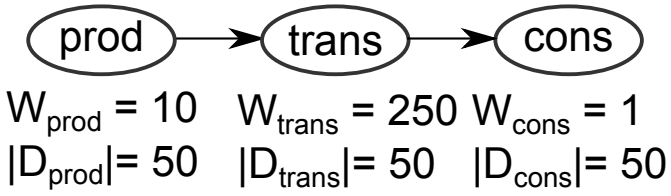

Figure 5.6: A PPN.

Figure 5.6 depicts a PPN with three processes. The prod process is a source process and the cons a sink process. All processes have an iteration domain size of 50, and are one-dimensional. The workload of prod is 10, the trans process has a workload of 250 clock cycles and for the cons process has a workload of 1 clock cycle.

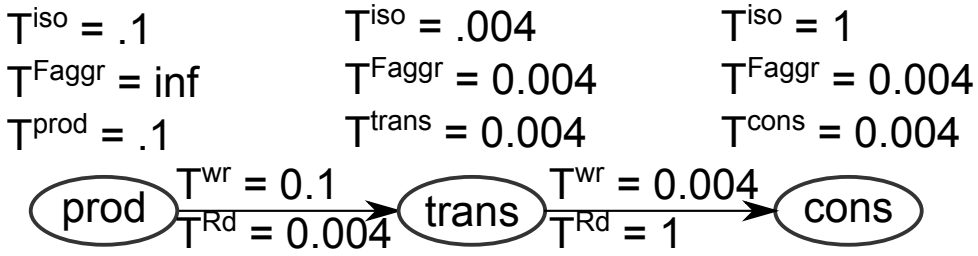

Figure 5.7: The throughput calculations of the PPN of Figure 5.6.

In Figure 5.7, the throughput is calculated for the processes and channels. Throughput calculation is explained in detail in Section 4.2.2.

The `Analyze` tool analyzes the different processes for split opportunities. The prod and cons processes are skipped by the tool, since these processes are sink or source processes. There is only one input argument consisting of one FIFO for process trans. This channel's write throughput is bigger than its read throughput. The read throughput is 1/25th of the write throughput. Therefore, the `Analyse` tool suggests to modulo split the trans process with factor 25.

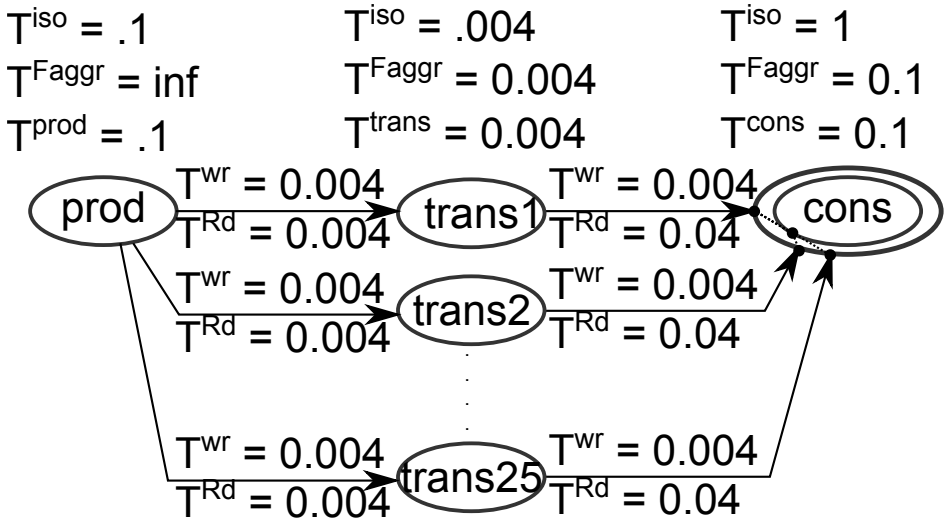

Figure 5.8: The PPN of Figure 5.6 split with a factor 25 by modulo split on trans as suggested by the analyze tool.

Figure 5.8 depicts the situation when split as suggested. Each partition of the trans process has an input channel from the prod process. The write throughput for each of these channels is different from the write throughput of the channel between prod and trans in the old situation. The write throughput is calculated as in Equation 4.4. The iteration domain of the prod process stays the same, as is the throughput of the prod process. The number of tokens designated for the channel is changed. Instead of all 50 tokens, only 2 are designated for each FIFO channel, as the tokens are distributed between all trans partitions. Since the write and read throughput for each of these channels is equal, the `Analyze` tool does not suggest any further split operations. The overall throughput of the program is improved, as all output channels of the trans partitions are connected to the same argument of the cons process. The aggregated throughput of cons is the sum of the throughput of all these channels. This result in an aggregated throughput of 0.1. The throughput of cons is now 0.1. The original program requires 12614 clock cycles to complete.

The transformed program only requires 805 clock cycles to complete.

5.3.2 Fork and Join Program

In this subsection the `Analyze` tool optimizes the PPN of Figure 4.4. This PPN is more complex than the PPN of the previous section as the PPN of Figure 4.4 contains a fork and a join process. The PPN consists of four processes. P1 is a source process, P4 a sink process. P1 has a workload of 10, P2 and P3 of 25 and P4 of 1. The iteration domain is one dimensional for all processes. P1 and P2 have an iteration domain of 50, P2 of 40 and P3 of 10.

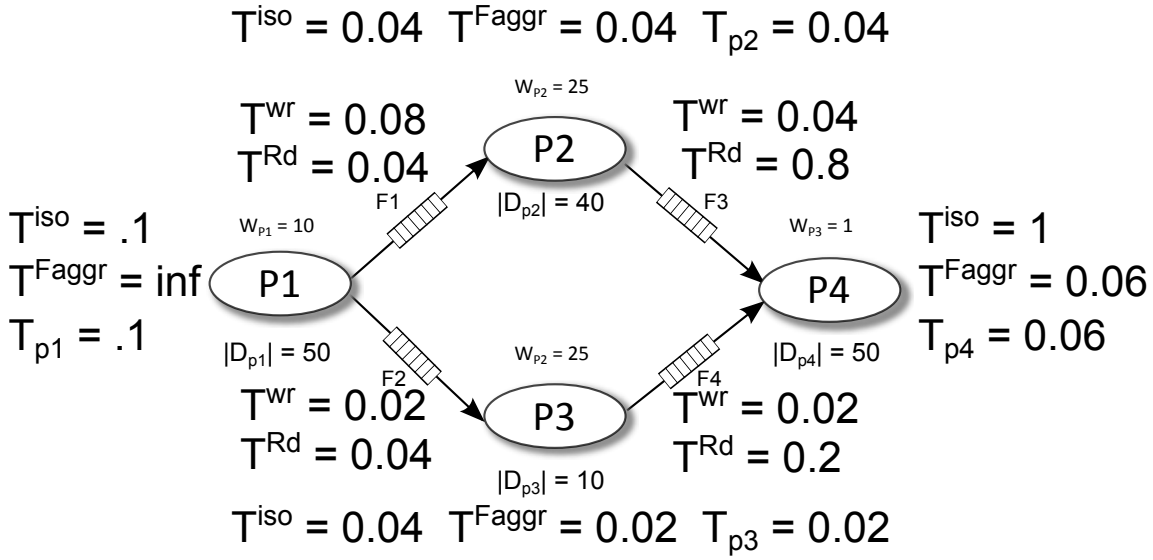

Figure 5.9: The PPN of Figure 4.4 with throughput calculations.

Figure 5.9 shows the PPN with throughput calculations. P1 and P4 can not be split as they are source and sink processes respectively. P3 has only F2 as input channel. This channel is examined for the decision how many times P3 should be split. The read throughput of F2 is higher than the write throughput. `Analyze` does not suggest a split action for P3, as the process is limited by write processes according to the throughput calculation. P2 has only channel F1 as input. The write throughput of F1 is 0.08 and the read throughput is 0.04. According the throughput calculations, P2 can be split by a factor 2. The `Analyze` tool concludes to split P2 with modulo split with factor 2.

After this split, the FIFOs of the P2 partitions have a different write throughput. Since only 20

of the 50 tokens from P1 goes to each partition, the write throughput of each partition's incoming FIFO is $20/50 \cdot .1 = 0.04$. The read throughput of these channels stays 0.04. The write throughput is not bigger than the read throughput and thus the `Analyze` does not suggest any more splits for these partitions. The program without transformations takes 1322 clock cycles to terminate. The transformed program takes only 793 clock cycles.

5.3.3 Sobel

In Section 5.2.1 the sobel application is described and depicted in Figure 5.1. In this section the `Analyze` tool is applied on the sobel application. In Figure 5.10, we see a part of the through-

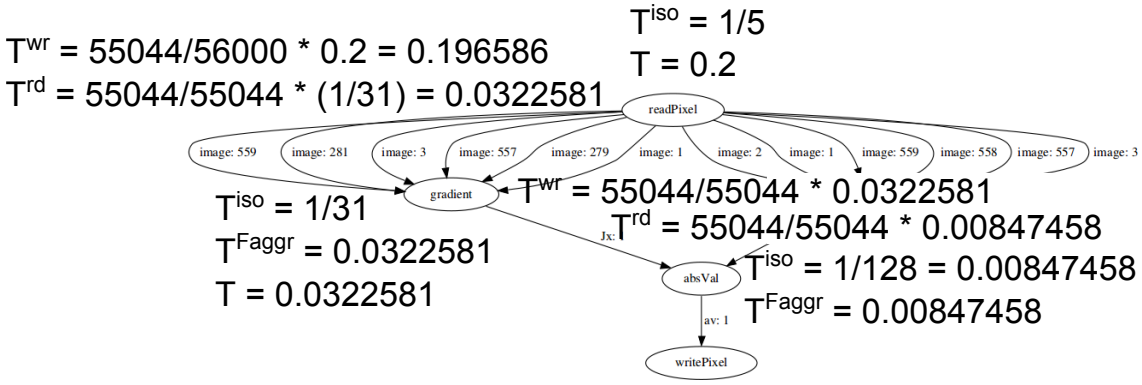

Figure 5.10: Sobel throughput calculation

put calculation for the sobel application. The throughput of the readPixel is equal to the isolated throughput since it is a source node, $T_{readPixel} = 0.2$. Each of the FIFOs from readPixel transport a total of 55044 tokens from the total of 56000 iterations readPixel produces a token. The write throughput for each of the FIFOs is $55044/56000 \cdot 0.2 = 0.196586$. Both gradient processes have an isolated throughput of $1/31 = 0.0322581$. The read throughput for all incoming FIFOs of the gradient processes is $T^{rd} = 55044/55044 \cdot (1/31) = 0.0322581$. The aggregated throughput and throughput of the gradient processes is $T_{gradient} = T^{Faggr} = T^{rd}$. Since all tokens a gradient process produces are transported over the only output FIFO the gradient node has, the write throughput of the FIFOs between a gradient process and the absVal process is $T_{gradient}$. the absVal process has an isolated throughput of $1/128 = 0.00847458$. The read throughput of the FIFOs between the absVal and the gradient nodes is 0.00847458 since each time absVal processes tokens it needs it from both channels. The aggregated throughput of absVal is 0.00847458.

The `Analyze` tool notices that the read throughput of the FIFOs between absVal and gradient processes is smaller than the write throughput of those FIFOs. The read throughput can be $T^{Wr}/T^{Rd} = 0.0322581/0.00847458 = 3.8$ times faster. The `Analyze` tool suggests to split the second dimension of the absVal node in four parts with the modulo-split method. The tool also suggests to split both gradient nodes seven times over the second dimension because all their incoming FIFOs also have a lower read throughput than write throughput and can be sped up at least $T^{Wr}/T^{Rd} = 0.196586/0.0322581 = 6.09$ times.

After applying the suggested changes, the throughput of the FIFOs between the gradient partitions and the readPixel process is changed. Now each FIFO has a write throughput of $7722/56000 \cdot 0.2 = 0.0275786$ or $7920/56000 \cdot 0.2 = 0.0282857$ depending on how many iterations the new gradient partition has. The read throughput of the FIFOs stays the same ($7722/7722 \cdot 0.0322581 = 0.0322581$ or $7720/7720 \cdot 0.0322581 = 0.0322581$).

While the absVal is split in four partitions, it now receives data from multiple gradient partitions for each argument. Each of the absVal partitions receives 1980 from the 12860 tokens from each gradient partition it is connected to. The read throughput of these FIFOs is $1980/13860 \cdot 0.00847458 = 0.00121065$, since this is smaller than the write throughput of those FIFOs it is also the throughput of the FIFOs. The aggregated throughput is the minimum of the input throughput per argument. On both input arguments a collection of FIFOs are sending data. The combined throughput for a argument is $0.00121065 \cdot 7 = 0.00847458$ as there are seven gradient partitions connected to one absVal partition per argument. The isolated throughput of this absVal partition is equal to the aggregated throughput.

When the `Analyze` algorithm analyzes the transformed version of the sobel application, it will suggest new transformations. The absVal partitions have an aggregated write throughput (minimum of combined write throughputs of all FIFOs connected to the same argument) of 0.0487929 and a minimal read throughput of 0.00847458. Thus, each absVal partition can be split $0.0487929/0.00847458 = 5.75755967$ times. There are four absVal partitions, thus the total suggested number of times the absVal partitions should be speed up is $5.75755967 \cdot 4 = 23.0302387$. Therefore, the `Analyze` tool suggests to merge all absVal partitions, and then split the resulting process in 24 parts over the second dimension by the modulo-split method. The tool does not suggest to split the modulo partitions in more pieces, as the write throughput of their incoming FIFOs is smaller than their FIFOs read throughput (0.0282857 or 0.0275786 as write throughput versus 0.0322581 read throughput).

When the previous suggestion is applied, the `Analyze` tool suggests to merge the absVal

partitions once more and split the resulting process in 28 partitions. At this point the whole system can keep up with the throughput of the source, which we do not transform, and the `Analyze` tool has no more transformation suggestions. Simulating the application where the gradient processes are split in seven pieces and the absVal process in 28 pieces, both split on the second dimension with the modulo-split method, requires 940688 clock cycles to execute completely. Comparing this with the findings of Section 5.2.1 we see that the result found by the `Analyze` tool is 85.89% faster.

5.3.4 Problems with certain conditions

Splitting Does Not Match Calculated Throughput Solution

Using the throughput algorithm of [11] as indicator what processes are potential candidates to split brings a number of problems. One of the limitations of the throughput algorithm is that the PPN should not contain cycles. Another problem is that by each split suggestion it is assumed the split will solve the read and write throughput difference between combined channels. In the examples of this section this worked perfectly, but this is not always the case. When it does not result in a solution where the read throughput of combined channels is equal or bigger as the write throughput, the next run of the `Analyze` tool would suggest a split in more partitions. This is not a problem, but it can take many `Analyze`,`Transform`,`pn` iterations before a solution satisfies the `Analyze` tool. The following example shows this problem.

We take as example the PPN of Section 5.3.1 (Figure 5.6). And change the workload value of the trans process to 200 clock cycles. The read throughput of the FIFO between the prod and trans is $50/50 \cdot 1/200 = 0.005$. The write throughput is 0.1. The `Analyze` tool suggests to split trans in 20 partitions. Unfortunately, the 50 iterations of trans cannot be equally divided among the 20 partitions, as each partition should contain $50/20 = 2.5$ iterations. The transformed PPN will have trans partitions with two iterations, and trans partitions with three iterations. The FIFOs between partitions with two iterations and the prod have a write throughput of $2/50 \cdot .1 = 0.004$. The read throughput is still 0.005.

In the next iteration of the `Analyze` tool, the `Analyze` tool will not suggest to split these processes as the read throughput is not smaller than the write throughput. The partitions with three iterations have incoming FIFO channels from prod with a write throughput of $3/50 \cdot .1 = 0.006$. The write throughput is bigger than the read throughput and the `Analyze` tool will suggest to split these partitions. The `Analyze` tool calculates that each of these partitions should be split

$0.006/0.005 = 1.2$ times. The `Analyze` tool reconstructs the original trans process by issuing a merge command. Accompanied with the merge command the tool gives a new split command. The split command splits the reconstructed trans process again in a number of partitions. The number of partitions is the sum of the times each partition should be split in, plus a factor one for each partition which does not need to be split. There are 10 partitions with 3 iterations. Each should be split 1.2 times. The 10 partitions should together be split in 12 parts. All 20 partitions should be split $1.2 \cdot 10 + 1 \cdot 10 = 22$ times. $50/22$ results in 8 partitions with 3 iterations, and thus the next run of the `Analyze` tool suggests another merge of all partitions and a split over 24 factors. When the `Analyze` tool examines the result it will merge and split again, by a factor 25 as it can not split by a factor 24.4. 50 iterations can be divided equally over 25 partitions, and the `Analyze` tool is done. Thus the optimization requires three `Analyze`, `Transform`, `pn` iterations before a satisfying result is found.

When the `Analyze` tool can not divide the iterations equal over the partitions the tool repeatedly split the partitions by increasing factors until the partitions are divided nicely.

A solution is to round each individual partition split factor up before taking the total sum of number of partitions to split on. In the example above, instead of splitting the reconstructed prod with factor $1.2 \cdot 10 + 1 \cdot 10 = 22$, the factor would be $\lceil 1.2 \rceil \cdot 10 + 1 \cdot 10 = 30$. This factor is too high, and results in partitions with 1 or 2 iterations. These partitions have a read throughput for their incoming channels higher than the write throughput for these channels. The `Analyze` tool will finish in fewer steps, at the cost of more processes in the final PPN.

Splitting and Merging Multiple Times

The current tool flow requires that the processes that are partitioned are merged before being split by the modulo-split method again. The current implementation can merge the partitions, but the resulting process keeps some unneeded sets constraints. These constrains do not contribute to the form of the set, they make the set more complex to deal with. After a number of merges and splits a set can be so complicated that the tools can no longer handle the set. This was for instance the case by the sobel application. We had to execute the splits on the original PDG instead of merging and splitting the PDG that was already been transformed before. This implementation problem could be overcome if the merged set could get rid of the non contributing constrains, or by recreating the original process in another way.

Chapter 6

Conclusions

In this thesis we have presented additional methods for designers to increase explicit parallelism in Polyhedral Process Networks derived from their programs. We have implemented these methods in two tools. The first tool created is the `Transform` tool. The tool splits selected nodes of Polyhedral Dependence Graphs. Polyhedral Dependence Graphs are translated into PPNs later in the tool flow, which results in a split process of a PPN if a node of the corresponding PDG was split by the `Transform` tool.

With these split methods we are able to make parallelism of an application explicit. Statements that could be run in parallel are placed in separate processes by splitting the process the statements are in.

the `Transform` tool performs a split by two different methods. The first method, plane splitting, divides the iteration set of a process based on a constant value for one or multiple dimensions. All iterations between two given values form a new process. The designer can determine the values of the planes.

The tool splits a process iteration domain in planes on a certain dimension on its own, given the number of partitions the process should be split into. The tool will search for split points which results in partitions with equal iteration domain sizes. The split can be executed in succession, splitting a process by multiple planes over multiple dimensions.

The second method is to split a process iteration domain with modulo splitting. A dimension of the domain is split based on a modulo factor. This method can also be applied in succession, splitting a process iteration domain over multiple dimensions by modulo factors. The tool can

merge previously split partitions of the same process, effectively undoing a previous split. These methods allow the designer to split a PPN to increase explicit parallelism. By placing each process on a different processor, the tool allows the designer to explore hardware configurations not possible to explore without creating explicit parallelism in the application. These hardware configurations in combination with the increased explicit parallelism in the application can improve performance.

The `Analyze` tool is able to improve the performance of PPNs automatically. Improvement is defined as decreasing the number of clock cycles a program requires to execute with no more resources than necessary. Resources are counted in number of processes.

The `Analyze` tool uses throughput calculations to determine how a PPN can be improved. The used throughput calculation method limits the class of PPNs possible to examine. As throughput can only be calculated if a PPN does not contain cycles, the `Analyze` tool can only handle PPNs without cycles. While the throughput algorithm gives a decent indication of what processes can be split, it also can incorrectly indicate it is useless to split certain processes. The split method the `Analyze` tool suggests is limited and only directed by how many times a process should be split. A suggested split does not always reach the improvements the tool assumes. Nonetheless, the tool is the first optimization tool capable of improving a PPN automatically. Despite its limitations, it is able to improve the PPN performance for a number of PPNs and can be run in iterative fashion to further optimize the transformed PPN. A designer could use the suggestions of the `Analyze` tool and execute these automatically or use them as guidelines for transformations the designer wants to apply himself. The designer can limit how fast the tool tries to optimize the PPN. This can be done by modifying the workload of the source processes.

The `Transform` tool is a robust tool for modifying the PPN. While some more transformations methods could be added, as is parameter support, the tool is satisfying the goal for basic splitting transformations. The designer has the option to specify exactly how a process should be split. The designer could also generally indicate with what method and in how many partitions a process should be split. This gives the designer the freedom how it indicates the detail of a splitting transformation.

The `Analyze` tool is limited by both the throughput model it uses and the simplified criteria it tries to optimize. While the tool is not perfect, it gives an indication of what processes can be split, and in a number of cases is able to find an improvement of the original PPN. It is a decent start for automated analysis of PPNs, but should be extended in the future to be more useful for complex cases the tool currently can not handle.

Chapter 7

Future Work

The `Transform` tool can split on modulo factors and planes. It can also merge previous split partitions of the same process. The ability to merge different processes would increase the possibilities for a designer to modify a PPN. Since the actual merging is only possible later in the tool flow, this was not possible to implement in the current tool. A solution should be found for this problem in the future. The automatic plane split method, which requires the size of the process iteration domain, can currently not handle domains specified by parameters, as the size of these domains is not expressible by an integer. An alternative method for automatically dividing a domain could be created, based on expressions containing symbolic constants.

The `Analyze` tool uses a number of simplifications for determining which processes to split. The balance between performance and resource cost should be possible to specify, instead of assuming that improvement is only based on execution performance of the PPN. The `Analyze` tool does also not take different types of communication channels into account. Channel delays are omitted in the current implementation. Typically channels between processes have a larger delay than self loop channels. Future work should take these different delay types into account.

The throughput calculations as used here have some limitations and do not always result in the correct solutions. Maximal Cycle Mean [15] calculations could give a more accurate indication for analysis, as well as being able to calculate values for PPNs that contain loops. To achieve this, the PPN should first be converted to a different graph [16].

The method used to split is currently limited to plane and modulo, where the method is selected by the designer and the tool does not examine the dependencies in detail for the best split direction

and dimension. This could be improved by examining the channels and connected processes before selecting the split method for each individual process that should be split.

Appendix A

Getting Started With The Tools.

A.1 Transform Tool

The `Transform` tool modifies PDGs in the tool flow as shown Figure 3.1. The tool requires a PDG without dependencies and command line input specifying which transformations should be executed. The transformation commands can either be provided by the designer or the `Analyze` tool. The tool gives as output the input PDG with the requested transformations applied on it.

A.1.1 Assumptions

The tool assumes the given PDG does not contain any dependencies between the nodes in the PDG. The nodes in the PDG are numbered from 0 and do not contain gaps i.e. after node nr 3 node nr 4 is defined, or none if node nr 3 is the last node. The nodes that are selected to be transformed are assumed not to have parameters. Some transformations might work with parameters but this is not tested, and thus the behavior is undefined.

A.1.2 Parameters

The syntax to call the `Transform` tool is

```
./pdgtrans (split command) < [file.yaml]
```

Where `(split command)` is a transformation command, and `[file.yaml]` the input PDG. `(split command)` is defined as

```
(split command) = (split type) (node number) (constraints) [(split command)]
                || (merge command) [(split command)]
```

`(split command)` can either be a split transformation or a merge transformation, followed by an arbitrary number of other `(split commands)`. The commands are executed in order from left to right.

`(merge command)` is a merge transformation. `(split type)` is the type of transformation that must be applied, `(node number)` the node on which the transformation applies, and `(constraints)` the parameters for the transformation.

The `(merge command)` is defined as follows

```
(merge command) = --union-merge --nodes (0-9)+,(0-9)+(,(0-9)+)*
```

Note that the merge command only works on nodes that are partitions of the same process. Attempting to merge different type of processes results in undefined behavior.
After `--union-merge --nodes` a sequence of comma separated node numbers should be given. These numbers correspond with the numbers nodes have in the PDG. At least two node numbers should be specified. The resulting node, representing the merged process, will have the first number of the node number sequence as its node number. This number can be used in other commands following the merge command.

```
(split type) =  --domain-split || --plane-split || --modulo-split
```

`--domain-split` splits a node by domains. The domain can be a list of isl sets. In this case each node created by the transformation is the selected node intersected with a given set. Conditions can be used instead of isl sets, in this case the resulting nodes are the original node intersected with a condition of the condition list.

`--plane-split` splits a node by planes. The planes can be given as conditions. The first created node by the transformation contains the elements below the first plane, the last node contains the elements above the last given plane, and all other nodes contain elements between two consecutive planes.

`--modulo-split` split a node by modulo factors. The selected node is split by a modulo factor for each required dimension. If more than one dimension has a split factor, the factors will be used in sequence, beginning with the highest dimension to split on. The node is first split on the highest of these dimensions. The partitions created by this transformation will be split by the modulo factor of the highest dimension not yet split on, this process continues until all given split factors are used.

The node number is given by the following syntax,

```
(node number) = --node (0-9)+
```

Where `(0-9)+` is the number of the node that should be transformed.

The `(constraints)` contain information how the transformation method should be applied on the selected node,

```
(constraints) = --conditions (conditions) ||
                             --sets (sets) ||
                        --factors (factors)
```

The transformations do not support all types of constraints. `--domain-split` transformation supports `--sets` and `--conditions`. `--modulo-split` supports `--factors`. `--plane-split` supports `--factors` and `--conditions`.

```
(comparison) = poly lib notation
(conditions) = "(comparison),(conditions)" ||
               "(comparison);(conditions)" ||
               "(comparison)"
(sets)       = "isl set notation;(sets)" || "isl set notation"
(factors) = fDim0[,Fdim1[,...[,FdimN]]]
```

`(factors)` is a list of comma separated numbers. The first number applies on the first dimension of the selected node, the second number applies on the second dimension of the node, etc. The values indicate for `--plane-split` in how many partitions a dimension must be split. Algorithm 1 is used to find the exact split points of the set.
The factor list tells the `--modulo-split` transformation with what factor each dimension must be modulo split. With both transformation methods a dimension can be skipped by giving it a 1 split value, and if multiple dimensions are given, the highest dimension is applied first, and each successive split value is applied on the created partitions.

`(sets)` is a list of semicolon separated isl sets. When using (sets) for the `--domain-split` method, the sets space names should match. i.e. splitting a node with a space name of `S_3`, the given sets to split this node on should also use as space name `S_3`.

`(conditions)` consists of a list of `(comparison)` separated by comma if the `(comparison)` should be added together as constraint, and separated by semicolon if the `(comparison)` are distinct conditions.

A `(comparison)` is a space-separated string of values. The first value indicates whether the comparison is an equality (0) or inequality (1). The following values represent the dimensions in sequential order. The last value represents the constant value. Parameters are not supported. As an

example the comparison `1 4 2 -3 -20` is interpreted as the inequality $4 \cdot dim1 + 2 \cdot dim2 + -3 \cdot dim3 + -20 >= 0$.

A.2 Transform Example Commands

In the following examples we use the sobel application as defined in Figure A.1

A.2.1 Splitting by Conditions

```
./pdgtrans --domain-split --node 1 --conditions
"1 1 0 -5;0 1 0 -5;1 1 1 -10;1 1 0 -5,1 1 1 -10" < ./sobel.yaml
```

Node 1 of the sobel PDG is split with the domain split method using conditions in polylib notation as input. Each partition receives the constraints assigned to it. A set constraints is given, separated by a comma to define more constraints for the same partition. A semi column is used to start a set of constraints for the next partition.

In this example four partitions are created as there are four condition sets (split by semi column). The first constraint set consists of one constraint, namely `1 1 0 -5`. The first value defines if the constraint is an equality (0) or inequality (1). The following values are conditions per dimension. Node 1 of sobel is a two dimensional set. The last value is the constant condition. In the example `1 1 0 -5` is the inequality $1 \cdot i + 0 \cdot j + -5 >= 0$ where i is the first dimension and j the second dimension.

Assume the iteration domain of node 1 was

```
S_1[i,j] : i >= 2 and i <= 199 and j >= 2 and j <= 279
```

The first partition will be

```
S_1[i,j] : i >= 5 and i <= 199 and j >= 2 and j <= 279
```

Where i must be bigger than 5 instead of 2. The second constraint is `0 1 0 -5`. This equality represents the constraint $1 \cdot i + 0 \cdot j + -5 = 0$. The partition with this constraint added will be `S_1[5,j] : j >= 2 and j <= 279`. Notice that the second dimension is now the constant value 5.

The third constraint is `1 1 1 -10`. The inequality is $1 \cdot i + 1 \cdot y + -10 >= 0$. The partition with this constraint added is

`S_1[i,j] : i >= 2 and i <= 199 and j >= 2 and j <= 279 and j >= 10 - i`. This constraint is dependent on two dimensions.

The last constraint set `1 1 0 -5,1 1 1 -10` consists of two constraints, namely `1 1 0 -5`

and `1 1 1 -10`. The resulting partition has an iteration set of
`S_1[i,j] : i >= 5 and i <= 199 and j >= 2 and j <= 279 and j >= 10 - i`. Note that both constraints are added. This method requires specific information how the partitions should look like. This can be unnecessary detailed in some occasions, but it also gives the designer a lot of control.

A.2.2 Splitting by Sets

```
./pdgtrans --domain-split --node 1 --sets
  "{ S_1[i,j] : i >= 5 }; { S_1[i,j] : j > 22}" < ./sobel.yaml
```

This command creates for each set an intersection with the original node. Two sets are defined, resulting in two partitions. The first partition is the intersection of the iteration set of node 1 with `{ S_1[i,j] : i >= 5 }` resulting in
`S_1[i,j] : i >= 5 and i <= 199 and j >= 2 and j <= 279`. The second set to intersect with is `{ S_1[i,j] : j > 22}`. The second partition created is
`S_1[i,j] : i >= 2 and i <= 199 and j > 22 and j <= 279`

A.2.3 Modulo Splitting by Factors

```
./pdgtrans --modulo-split --node 1 --factors 3,4  < ./sobel.yaml
```

Splits node 1 in $3 \cdot 4$ partitions, where each of these new nodes is a unique offset combination of modulo 3 for the first dimension and modulo 4 for the second dimension. One of the partitioned sets is

```
{ S_1[i, j] : exists (e0 = [(-3 - j)/4], e1 = [(-1 + i)/3]:
  4e0 = -3 - j and 3e1 = -1 + i and i >= 2 and i <= 199
  and j >= 2 and j <= 279) }
```

This partition contains all iterations where the first dimension values modulo 3 result in 1 and second dimension values modulo 4 in 1.

A.2.4 Plane Splitting by Conditions

```
./pdgtrans --plane-split --node 1 --conditions
  "0 1 0 -5,0 1 0 -10,0 1 0 -15" < ./sobel.yaml
```

With this command node 1 will be split using a number of planes. Comparisons are separated by a comma. Each comparison represents a plane. Every iteration with a value under the first plane

is in the same partition. All iterations with a value equal or higher of the first plane and lower than the second defined plane are in the second partition. All iterations with a value equal or higher than the last defined plane are in the last partition. For this split operation it is assumed that all defined planes are in order of lowest values to highest values and planes do not intersect. If these two assumptions are not met undefined behavior can occur. In this example three planes are defined. The first is $1 \cdot i + 0 \cdot j - 5 = 0$ which is $i = 5$. Therefore, the first partition will contain all iterations where the value of i is less than 5: `{ S_1[i, j] : i >= 2 and i <= 4 and j >= 2 and j <= 279 }`. The second plane is defined as $i = 10$. All iterations of node 1 with a value of i between 5 and 10 (including 5, excluding 10) are placed in the second partition, i.e.
`{ S_1[i, j] : i >= 5 and i <= 9 and j >= 2 and j <= 279 }`. The third partition contains all iterations of node 1 between the second and third defined plane, including the values of the second plane. The third plane is $i = 15$ and the third partitions iteration set is
`{ S_1[i, j] : i >= 10 and i <= 14 and j >= 2 and j <= 279 }`. The last partition contains all iterations with a value equal or above the last plane i.e.
`{ S_1[i, j] : i >= 15 and i <= 199 and j >= 2 and j <= 279 }`

A.2.5 Plane Splitting by Factors

```
./pdgtrans --plane-split --node 1 --factors 2,3 < ./sobel.yaml
```

Splits node 1, by dividing the set in two parts over the first dimension. Then, each of these parts are split by dividing them in three parts over the second dimension. The splits are done by splitting on one dimensional planes. While the algorithm tries to split each set in equal parts, giving more than one dimension might result in unequal parts. A dimension can be skipped for splitting, by giving it a factor 1, meaning one part, therefore no split on this dimension. This example results in $2 \cdot 3$ partitions. They are

```
S_1[i, j] : i >= 2 and i <= 100 and j >= 2 and j <= 94
S_1[i, j] : i >= 2 and i <= 100 and j >= 95 and j <= 187
S_1[i, j] : i >= 2 and i <= 100 and j >= 188 and j <= 279
S_1[i, j] : i >= 101 and i <= 199 and j >= 2 and j <= 94
S_1[i, j] : i >= 101 and i <= 199 and j >= 95 and j <= 187
S_1[i, j] : i >= 101 and i <= 199 and j >= 188 and j <= 279
```

Node 1 is split in 6 partitions, where i is split in two parts and j in three parts.

A.2.6 Merging Partitions

```
./pdgtrans --union-merge --nodes 2,5,7,12 < ./someFile
```

Merges nodes 2, 5, 7, and 12 into one node, with number 2. Note that the nodes must all have one original node as parent. Thus merging can only take place if the nodes are partitions of the same original process. After a merge the node in the list named first is the number the merged node is named. This number can then be used to refer to the merge node for additional commands. Note that after all commands are processed the node numbers will be renumbered such that the PDG contains a sequence of node numbers without gaps. E.g. take a PDG with nodes 1 till and including node 20. if the previous example would be executed, during the tool execution all nodes keep their original number, except the nodes specified in the merge command, as they are removed and a new merged node nr 2 will be in their place. After all commands are processed, gaps will be removed. There are gaps as nodes nr 5,7 and 12 no longer exists. All nodes after node nr 4 will be decreased one in number value, to fill the gap of node nr 5. All nodes with a number higher than 6 will be decreased by one additional value, to also fill the gap of node 7. Nodes with an original number above twelve will have a new number value three lower than their current value, such that also the gap of node nr 12 is filled. Node nr 20 will have a new value of 17, and node nr 6 will have a new value of 5 for instance.

A.2.7 Analyze Tool Use and Syntax

The `Analyze` tool requires a PDG with dependencies and optional designer input and returns a list of commands in the correct syntax for the `Transform` tool to split the PDG without dependencies. See Figure 4.1 for the location of the `Analyze` tool in the tool flow.

The PDG the `Analyze` tool receives as input should contain dependencies created with the `pn` tool. The PDG should not contain any cycles and self loops. The PDG should also not contain any parameterized sets, as these sets do not have an absolute size. In case the PDG is split by modulo, the tool assumes the iteration sets do not contain any gaps, unless split by a previous command from the `Analyze` tool. The sets could still contain gaps, but the tool will not attempt to adjust the split command to spread the gaps evenly between the potential partitions.

The command to execute the program is as follows:

```
./pdganalysis < fileWithDependiciespn.yaml (command)
(command) = (--method=modulo || --method=plane || --round) [(command)]
```

`--method=modulo` and `--method=plane` indicate which split method the tool returns. Only one of the two can be given, the default is the modulo method. In case of the modulo method the tool returns modulo splits with factors. If the tool is used on a PDG already split before, the returned

split suggestion could include merge commands for partitions of the same stem process. When the plane method is activated, the set will be suggested to split by splitting on one-dimensional planes. Since the returned command uses the plane split by factors method of the `Transform` tool, this method most likely results in equally distributed partitions even when the partitions contain irregular gaps. A disadvantage as described in Section 4.2.5 is that partitions process data in sequence, it is most likely the speedup the tool assumes it creates is not actually achieved. `--round` is used in combination with the modulo split method to limit the number of `Analyze` needed to come to a stable point, with as trade off a more resource-wasting solution, as discussed in Section 5.3.4.

The `Analyze` tool returns a command interpretable by the `Transform` tool. The PDG version with dependencies is used by the `Analyze` tool, the same PDG without dependencies by the `Transform` tool. The returned PDG from the `Transform` tool can be used by the `pn` tool to add dependencies. This PDG can then be used by the `Analyze` tool to find a better solution for the PDG. This sequence can be repeated multiple times. The `Analyze` tool returns nothing in case the input PDG is optimal according the throughput calculations of the `Analyze` tool.

The tool sequence `Analyze`, `Transform` and `pn` can be programmed in a script to execute this sequence with ease.

```
int main(void)
{
  int i, j;

  static int image[1000][1000];
  static int Jx[1000][1000];
  static int Jy[1000][1000];
  static int av[1000][1000];


  for (i=1; i <= M; i++) {
    for (j=1; j <= N; j++) {
      readPixel(&image[i][j]);
    }
  }

  for (i=2; i <= M-1; i++) {
    for (j=2; j <= N-1; j++) {
      gradient( &image[i-1][j-1], &image[i][j-1], &image[i+1][j-1],
        &image[i-1][j+1], &image[i][j+1], &image[i+1][j+1], &Jx[i][j] );
    }
  }

  for (i=2; i <= M-1; i++) {
    for (j=2; j <= N-1; j++) {
      gradient( &image[i-1][j-1], &image[i-1][j], &image[i-1][j+1],
        &image[i+1][j-1], &image[i+1][j], &image[i+1][j+1], &Jy[i][j] );
    }
  }

  for (i=2; i <= M-1; i++) {
    for (j=2; j <= N-1; j++) {
      absVal( &Jx[i][j], &Jy[i][j], &av[i][j] );
    }
  }

  for (i=2; i <= M-1; i++) {
    for (j=2; j <= N-1; j++) {
      writePixel( &av[i][j] );
    }
  }

  return 0;
}
```

Figure A.1: Top-level C code of the sobel application.

Acknowledgements

For the realization of this thesis I would like to thank the following people. First, my supervisors drs. Sven van Haastregt, dr. ir. Todor Stefanov and second reader dr. Jetty Kleijn, for their support. In particular drs. Sven van Haastregt for the time he spend to guide me throughout this project. Second, the people of the LERC group for testing the tools and their feedback. And last but not least, my family for their support throughout my study.

Bibliography

[1] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[2] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[3] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM.

[4] Rainer Domer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and DanielD Gajski. System-on-chip environment: A specc-based framework for heterogeneous mpsoc design. *EURASIP Journal on Embedded Systems*, 2008(1):647953, 2008.

[5] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith. Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models. In *Proceedings of the 45th annual Design Automation Conference*, pages 580–585. ACM, 2008.

[6] J. Falk, J. Gladigau, M. Glaß, C. Haubelt, S. Helwig, J. Keinert, M. Lukasiewycz, T. Schlichter, T. Streichert, M. Streubühr, et al. Systemcodesigner–the system-level hardware-software-co-design tool.

[7] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. Efficient symbolic multi-objective design space exploration. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 691–696. IEEE Computer Society Press, 2008.

[8] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: an integrated framework for mpsoc application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008.

[9] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal. Multi-processor system-level synthesis for multiple applications on platform fpga. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 92–97. IEEE, 2007.

[10] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 7–12. ACM, 2002.

[11] S. Meijer et al. *Transformations for polyhedral process networks*. Ph.D. Thesis. Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2010.

[12] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[13] E. Cannella, O. Derin, and T. Stefanov. Middleware approaches for adaptivity of kahn process networks on networks-on-chip. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1–8. IEEE, 2011.

[14] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, et al. *Numerical recipes*, volume 547. Cambridge Univ Press, 1986.

[15] S. Sriram and S.S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*, volume 3. CRC, 2009.

[16] H.N. Nikolov and T.P. Stefanov. Analytical performance evaluation of streaming applications modeled as sdf, csdf, and ppn models of computation. LIACS Technical Report 2012-03, March 2012.