Transformations for polyhedral process networks

Meijer, S.

Chapter 2

Background

In this chapter, we give the definitions and notations that are used throughout the rest of this dissertation, i.e., we review some basic mathematical notations and definitions as discussed in for example [72, 74]. We thereby focus on polyhedra and the polyhedral model that are used by compiler optimizations to efficiently analyze and transform input programs. Then, we define the input programs, i.e., the class of applications, that can be analyzed with this polyhedral model and show an example of a Polyhedral Process Network (PPN). We discuss the structure and properties of PPNs, which is necessary to understand the chapters that deal with analyzing and transforming PPNs.

2.1 Polyhedra

The **scalar product** or **inner product** of two vectors **a** and **b**, denoted by $\mathbf{a} \cdot \mathbf{b}$, is defined as $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \sum_{i=1}^{n} a_i b_i$, where $a = (a_1, .., a_n)$ and $b = (b_1, .., b_n)$ are column vectors. Note that $\mathbf{a} \cdot \mathbf{b} = \mathbf{0}$ iff vectors **a** and **b** are orthogonal or $\mathbf{a} = \mathbf{b} = 0$.

Given a non-zero vector **y** in $\mathbb{R}^n$ and a constant α, the following sets of points are defined:

- A **hyperplane** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} = \alpha\}$.
- A **closed half-space** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} \geq \alpha\}$.
- An **open half-space** $H = \{\mathbf{x} \mid \mathbf{x} \cdot \mathbf{y} > \alpha\}$.

An affine hyperplane is a $(d-1)$-dimensional hyperplane in a d-dimensional space, and thus divides the space in exactly two parts. A line, for example, is an affine

hyperplane in a 2-dimensional space, but not in a 3-dimensional space. We will use hyperplanes to define a polyhedron, but also in the process splitting transformation to partition processes in PPNs (see Chapter 3).

A **rational polyhedron** $\mathcal{P}$ is a subset of $\mathbb{Q}^d$ bounded by a finite number of closed half-spaces, i.e.,

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq \mathbf{b}\} \tag{2.1}$$

where A is an integral $m \times d$ matrix, and b is an integral vector of size m.

A **polytope** is a bounded polyhedron.

Figure 2.1 shows two 2-dimensional spaces with a number of closed half-spaces defining two polyhedra. The purpose of this example is to show the difference between a polyhedron and polytope. In Figure 2.1 A), a polyhedron is shown that is defined by only two constraints. As a result, the polyhedron is unbounded because there are no constraints on the maximum values that the points can have. In contrast, Figure 2.1 B) shows 4 lines/constraints that encapsulate all points within the grey area, which makes it an example of a bounded polyhedron, i.e. a polytope.

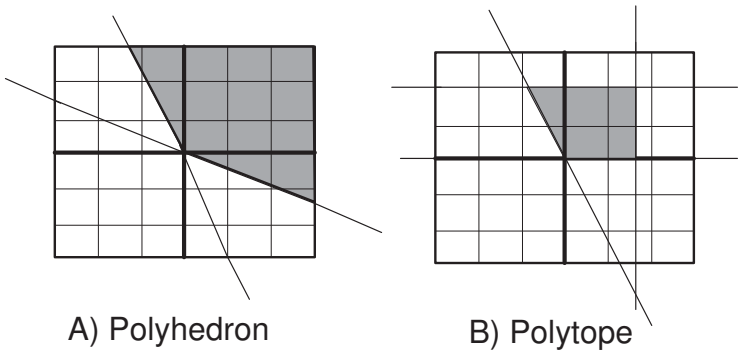

Figure 2.1: Polyhedron vs. Polytope

Polyhedra can also depend on a vector of parameters, denoted by $\mathbf{p}$, and we therefore define a parameterized polyhedron, denoted by $\mathcal{P}(\mathbf{p})$.

A parameterized polyhedron $\mathcal{P}(\mathbf{p})$ is a polyhedron whose closed half-spaces are affinely dependent on a vector of parameters $\mathbf{p} \in \mathbb{Q}^d$, i.e.,

$$\mathcal{P}(\mathbf{p}) = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq B\mathbf{p} + \mathbf{b}\} \tag{2.2}$$

where A is an integral $m \times d$ matrix, B is an integral $m \times n$ matrix, and b is an integral vector of size m.

We use polyhedra to model all iterations of a program statement in nested-loop programs. That is, we extract and use the polyhedral model to efficiently analyze and transform input programs, which we further discuss in Sections 2.4 and 2.5. In Section 2.2, we first discuss how different points in a set can be compared and ranked using Parametric Integer Linear Programming (PILP) techniques.

2.2 Lexicographic Order

In program analysis, many problems can be formulated as a Parametric Integer Linear Programming (PILP) problem. An example of such a problem is to find the first, or last, array element accessed by a program statement in a nested-loop. Thus, parametric integer programming [24], [74] is used to find exact solutions and feasible points ranked according to a lexicographic order. In program analysis of nested-loop programs, we are dealing with sets of integer vectors defined by linear inequalities. If we consider a set S as an example, then recall from Section 2.1 that it is defined as $S = \{x \in \mathbb{Z}^d \mid Ax \geq b\}$ with $A \in \mathbb{Z}^{m \times d}$ and $b \in \mathbb{Z}^d$. Then, parametric integer linear programming is used to find the minimum or maximum point in set S. And two points $a \in \mathbb{Z}^n$ and $b \in \mathbb{Z}^n$ in set S can be compared by using the lexicographic order.

We say that **a** is **lexicographically smaller** than **b**, denoted by $\mathbf{a} \prec \mathbf{b}$, if for the first position i in which both vectors are different, we have $a(i) < b(i)$. This is expressed as a set of equalities and inequalities as:

$$\mathbf{a} \prec \mathbf{b} \equiv \bigvee_{i=1}^{n} \left(a(i) < b(i) \wedge \bigwedge_{j=1}^{i-1} a(j) = b(j) \right) \tag{2.3}$$

Let us take as an example a set S with 5 elements: $S = \{(1,1), (1,2), (2,1), (2,2), (2,3)\}$. Using Formula 2.3, we see that $(1,1)$ is lexicographical smaller than $(1,2)$, denoted by $(1,1) \prec (1,2)$, because $(1 = 1 \wedge 1 < 2)$. Similarly, we see that $(1,1)$ is lexicographical smaller than $(2,3)$, i.e., $(1,1) \prec (2,3)$, because comparing the first component of both points gives $(1 < 2)$. Element $(1,1)$ is the smallest element of set S and we define it as the lexicographical minimum element, denoted by $lexmin$. Similarly, we also define the lexicographical maximum point as the largest element, denoted by $lexmax$. For set S, element $(2,3)$ is the largest element. The problem of finding the lexicographical minimum/maximum point within a set of linear constraints can be solved with PILP. The example set S as we have defined it above

can also be represented by a set of constraints, i.e., $S = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq 2 \wedge 1 \leq j \leq 3\}$, and the ILP problem (no parameters are used in this example) can be subsequently formulated as shown in Table 2.1.

Objective:	$lexmin\{(i,j)\}$
Subject to:	$1 \leq i \leq 2$
	$1 \leq j \leq 3$

Table 2.1: Constraint system

The solution to find the minimum point for a given convex domain is based on the dual simplex algorithm [48] that is implemented in open-source libraries such as isl [93], Parma Polyhedral Library [4], and Piplib [24]. On a very high-level, the idea of the PIP algorithm and dual simplex method, is to find a minimum real point for a given convex set. Then, iteratively, new constraints are added not removing any integer points from the set. These libraries will thus find $(1,1)$ as the lexicographical minimum, and $(2,3)$ as the lexicographical maximum point.

Using the lexicographical order, it is also possible to rank an iteration point in polyhedra.

Definition 1 *The **rank** of a point $p \in \mathcal{P}$, is a number $n \in \mathbb{Z}$ denoting all points that are lexicographical smaller than p.*

For example, let us consider point $(i = 1, j = 3)$ of the `filter` function call statement in Figure 2.3 B). To rank this point, we use the lexigraphical order to determine all points that precede $(1,3)$. Therefore, we first consider all points that are smaller in the first component of point $(1,3)$, i.e., $i < 1$. The points that satisfy this constraint, corresponds to all points within the top most and largest grey box in Figure 2.3 B); for all these points $i = 0$. In addition, we consider the points that have the same value in the first component, but which have a smaller value in the second component, i.e., $i = 1 \wedge j < 3$. This corresponds to all points within the second and smallest grey box in Figure 2.3 B). Thus, the rank of point $(1,3)$, corresponds to the number of elements in the set $(i < 1 \ \vee \ (i = 1 \wedge j < 3))$, i.e., all greyed points in Figure 2.3 B). If we assume $N = 100$, then the rank of $(1,3)$ is $100+2 = 102$, which is thus obtained by counting the number of points in a set. Counting the number of points in (parametric) polyhedra, i.e., the enumeration of (parametric) polyhedra, is a research field in itself. The basic idea is to derive a quasi-polynomial that describes the number of integer points in a polytope $\mathcal{P}$. For an in-depth discussion, the reader is referred to, for example, the works [18], [97]. In this dissertation, we use that work which is implemented in the polyhedral library PolyLib [98]. Thus, when we want

to know the cardinality, or the number of points, of a set S, which we denote by $|S|$, then we use the counting functions from these libraries.

2.3 Static Affine Nested-Loop Programs

In Section 2.5, we consider parallel application specifications that are functionally equivalent to sequential program specifications that are static affine nested-loop programs. These are the subject of this section.

Definition 2 *A **static affine nested loop program (SANLP)** is a program where each program statements is enclosed by one or more loops and if-statements, and where:*

- *loops have a constant step size;*
- *loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;*
- *if-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;*
- *index expressions of array references are affine constructs of the enclosing loop iterators, static program parameters, and constants;*
- *data flow between statements in the loop is explicit, which prohibits that two statements that contain function calls communicate through shared variables invisible to the compiler.*

An example of a static affine nested-loop program is shown in Figure 2.2.

```
 1  #parameter  10 <= N <= 100;

 2  for (i=0; i<= 2*N; i++)
 3    for (j=0; j<= 4*N; j++)
 4       a[i][j] = read_data ();          // statement S0

 5  for (i=0; i<= N; i++) {
 6    for (j=i; j<= N; j++) {
 7      if (i+j <= N-1) {
 8        a[i][j] = filter(a[2*i][4*j]);  // statement S1
 9      }
10      write_data(a[i][j]);              // statement S2
11    }
12  }
```

Figure 2.2: Example code of a SANLP

A static program parameter `N` is defined in `line 1`. This static parameter indicates that `N` can take a value between 10 and 100 which, however, cannot change at run-time. Using static parameters is very useful because an equivalent parallel specification, such as a PPN, needs to be derived only once, even if some requirements of the application change. Loops need not necessarily be perfect nests. That is, the program statements can appear at any level of the nested-loop, and thus not necessarily at the innermost loop level. Furthermore, the program statements can be guarded by if-statements, as shown in `line 7`. However, the conditions in these if-statements can only be affine combinations of loop iterators, static program parameters, and constants, and thus cannot have data dependent behavior. The functions in `line 4, 8, 10` read and write data only through arrays, and not for example through shared variables, or pointers to the arrays not visible to the compiler. In other words, the data flow is made explicit by reading/writing data only through affine array accesses.

The polyhedral model is an appealing model to represent and manipulate loop nest structures and their program statements in static affine-nested loop programs, as shown in for example [69], [63], [70]. Program parts that can be modeled with the polyhedral model are called *static control parts (SCoPs)* in the compiler community [76]. To be more precise, a SCoP is defined as a single-entry-single-exit region of the control-flow where loops bounds and conditional predicates are affine functions enclosing loop counters and invariant parameters. Once the polyhedral model is extracted from a SANLP or SCoP, see Section 2.4, data dependence analysis and loop restructuring transformations such as loop fusion, loop fission and strip-mining can be efficiently implemented using existing tools (e.g., PolyLib [98], the Parma Polyhedral Library, and Cloog [7]). The reason is that the iteration domain of a program statement, i.e., all iterations of that statements, are represented by a single geometrical object - a polyhedron. This polyhedron can be analyzed with PILP techniques as presented in Section 2.2.

Although the polyhedral model does impose some restrictions on the input program, in many application domains it is natural to express time critical parts of the applications in the form of a SANLP. Examples are DSP and audio/video stream-based applications in consumer electronics, modeling and simulation applications in high performance computing, molecular biology, radio astronomy, medical imaging, and high energy physics. Therefore, the polyhedral model is highly relevant because it enables efficient code restructuring and analysis in many program code parts.

2.4 Extracting the Polyhedral Model from SANLPs

The polyhedral model is a description of all program statements and their iteration points in Static Affine Nested-Loop Programs (SANLPs) with polyhedra. We refer to all iteration points of a program statement as the iteration domain, which in the program code (i.e., the SANLP) is defined by the enclosing loops of the program statements. Since the iteration points of a program statement are executed in a particular order, the polyhedral objects that model these iterations are ordered as well, i.e., the polyhedral model that we use for our program analysis consist of:

- polyhedra that define the iteration domains of program statements,
- a lexicographical ordering (see Section 2.2) of the points within the polyhedra.
- and data access functions for array references, which map a point from the iteration domain to a point in the data space that is accessed by the array references, i.e., the affine index expression as discussed in Section 2.3.

In the polyhedral model that we extract from SANLPs, an iteration vector is associated with each program statement. The dimension of the vector is equal to the number of loops that enclose the statement. The i-th component of the vector corresponds to the value of a loop iterator at depth i. Thus, the iteration domain of a statement is given by a set of linear inequalities defining a polyhedron in an d-dimensional domain, where d corresponds to the dimension of the iteration vector, i.e., the depth of the enclosing loop nest. In fact, the polyhedral model of the iteration domain of a statement is just a set of linear equalities and inequalities. Here is an example.

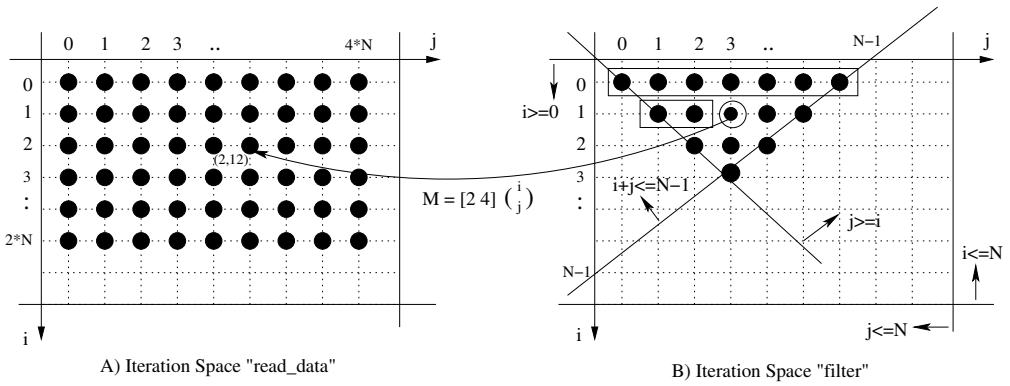

Figure 2.3: Iteration Space of $read\_data$ and $filter$ Function Call Statements

Figure 2.3 shows the two iteration domains of the `read_data` and `filter` function call statements from Figure 2.2. Let us focus on the iteration domain of the

`filter` function call statement shown in Figure 2.3 B). For brevity, we refer to this statement as $S1$. Since statement $S1$ is enclosed by two for-loops i and j, its iteration domain is 2-dimensional, and is referred to as D_{S1}. The lower/upper bounds of the enclosing loops are the first constraints that we take into account when defining the iteration domain of $S1$. Loop i starts at 0 and has maximum value of N, which translates to the following 2 constraints: $i \geq 0$ and $i \leq N$. Loop j has an initial value equal to i and has a maximum value of N, which translates to another two constraints: $j \geq i$ and $j \leq N$. In addition to the constraints imposed by the lower/upper bounds of loops, the execution of program statement $S1$ is guarded by an if-statement, which imposes another restriction on the iteration domain, i.e, only iteration points smaller than $i + j \leq N - 1$ are executed. Figure 2.3 B) shows 5 different lines in a 2-dimensional domain, which correspond to the 5 constraints imposed by the upper/lower bounds of the loops and the if-statement as we have described above. Thus, the constraints restrict the iterations points that are executed by $S1$, and the iteration points actually executed by $S1$ are denoted by the solid dots in Figure 2.3 B), i.e., they form a triangle. These iteration points are executed in the order from top to bottom and from left to right. We have extracted all constraints on the execution of $S1$ to define its iteration domain D_{S1} in the polytope representation: $D_{S1}(N) = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq N \wedge i \leq j \leq N \wedge i + j \leq N - 1\}$. All executions of program statement $S1$ are in this way represented with one geometrical object, i.e., a polytope. Once an iteration domain has been extracted for a statement, it can be efficiently further analyzed and transformed using polyhedral analysis and tools. For example, the number of integer points of an iteration domain can be counted [18], [96] which is useful for loop optimizations [20] and data cache analysis [19]. Another application is the (re)scheduling of iterations and subsequently the code generation of iteration domains [7].

2.5 Polyhedral Process Networks

Extracting the polyhedral model for SANLPs as discussed in Section 2.4, enables exact data-flow analysis of scalar and array references. This exact data flow analysis uses the PILP techniques as discussed in Section 2.2 and is the most fundamental step in deriving PPNs in a fully analytical way from SANLPs as described in [89, 90, 95]. For an in-depth discussion on the derivation of PPNs, the reader is referred to these works. In this section, we only discuss the different properties of PPNs, and show the corresponding PPN for the code example in Figure 2.2.

In the partitioning strategy of the `pn` compiler [95], one autonomous process with local control and memories is created for each program statement. Subsequently, the control for the FIFO communication is automatically derived. We refer to pro-

cess networks derived by the `pn` compiler as *polyhedral process networks* (PPNs). The reason is that they are functionally equivalent to Static Affine Nested Loop Programs (SANLPs), the processes are structured in a particular way, and the execution of processes and FIFO reads/writes are described by polyhedra. Polyhedral process networks are, therefore, a special case of Kahn Process Networks (KPNs) [40], because Kahn Process Networks is a simple, yet powerful model of computation that only specifies how processes synchronize and communicate. Thus, the KPN model of computation does *not* impose any restrictions on, for example, the internal structure of processes and only defines that processes use a blocking FIFO read primitive and have unbounded FIFO buffers. However, as already mentioned above, the processes in PPNs are internally structured in a particular way. That is, in each execution of a process, we can distinguish a *Read* phase (R), an *Execute* phase (E), and a *Write* phase (W). To be more specific, a process consists of:

1. a list of *input port domains* to read all the function input arguments from the corresponding input FIFO channels,

2. a *function* that processes the input arguments and produces function output arguments, and

3. a list of *output port domains* to write the function output arguments to the corresponding output FIFO channels.

There can be two exceptions: *source* and *sink* processes. The former only generates data and does not read any data from other processes. The latter only collects data and does not write any data to other processes. However, source/sink processes can have incoming/outgoing channels, but then these channels are self-channels and data is read/written from/to itself. We illustrate the structure of the processes in a PPN with an example shown in Figure 2.4. This PPN is derived from the SANLP shown in Figure 2.2, where we have set the parameter N to 100. Since that SANLP consists of 3 statements $S0$, $S1$ and $S2$, the corresponding PPN consists of 3 processes $P0$, $P1$ and $P2$.

It can be seen that process $P0$ is a source process because it does not read data from other processes, and that process $P2$ is a sink process because it does not write data to other processes. Process $P1$, on the other hand, first reads data from FIFO channel $F1$, processes it by executing function `filter`, and writes the result to its outgoing FIFO channel $F3$. Thus, it clearly shows the different read, execute, write phases as also indicated with the letters R, E, and W in Figure 2.4. Furthermore, we see that each process executes a particular function that corresponds to a function from the SANLP.

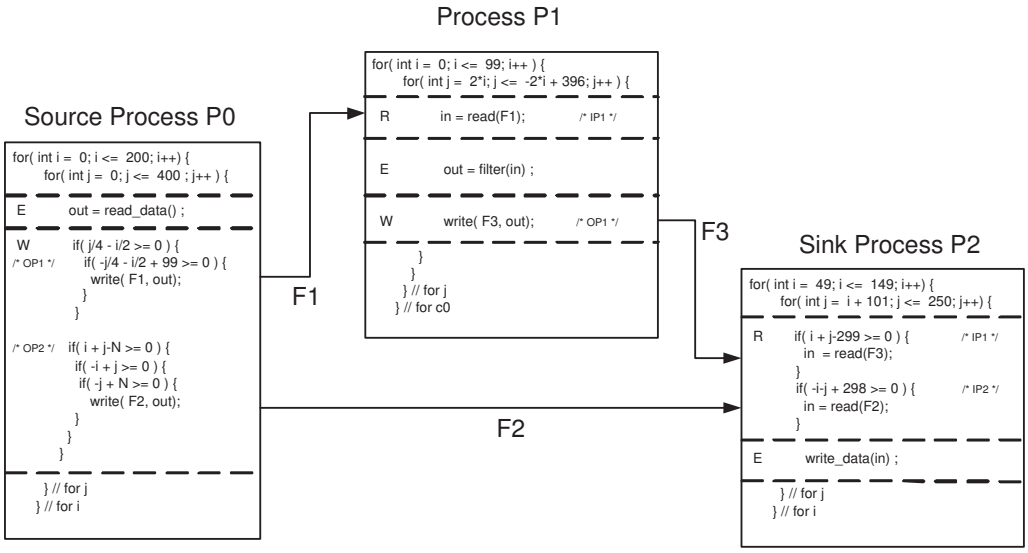

Figure 2.4: Derived Polyhedral Process Network (PPN) Model and its Representation in Executable Program Code

Definition 3 *A **process function** represents the computational part of a process. It corresponds to a function call statement in the sequential application that is a pure function without side-effects which only reads/writes through its input/output arguments.*

For the PPN in Figure 2.4, the process function of $P0$ is `read_data`, and we see that `filter` and `write_data` are the process functions of processes $P1$ and $P2$, respectively. These process functions are important for the process splitting and merging transformations that we present in Chapters 3 and 4, because the goal of both transformations is to create a more load-balanced PPN. Therefore, it is important to know the cost for executing the process function once, and we refer to this as the process workload.

Definition 4 *The **process workload** of a process P_i, denoted by W_{P_i}, represent the total number of required time units to execute the process function once, provided that i) all its input data is available (i.e., the reading phase is ignored), and ii) the time to write the output data is excluded (i.e., the writing phase is ignored).*

To give an example for the PPN shown in Figure 2.4, one can think of the `read_data` process function as a very light-weight process function that only reads data from a

memory location, i.e., the process workload W_{P0} is very small as executing that function is completed in a few clock cycles. The `filter` process function can be considered to be a more coarse-grain function as some actual computation is performed on the data. Thus, the process workload W_{P1} is much larger than W_{P0}. Similar to the `read_data` function, the `write_data` process workload W_{P2} is very small as only data is written back to some memory location and not any computations on the data is performed. The process workload does not include the time required to read/write the data before/after executing the process function.

Definition 5 *A **process iteration** of a process P_i is defined as a single execution of the process function, where first all input data is read from incoming FIFO channels (i.e., the read phase), the process function is executed (i.e., the execute phase), and subsequently all output data is written to outgoing FIFO channels (i.e., the write phase).*

All iterations of a process are described by a process iteration domain.

Definition 6 *The **process iteration domain** of a process P_i, denoted by D_{P_i}, is defined as all process iterations of process P_i and is described by a set of equalities and inequalities, i.e., a polytope.*

Thus, the process iteration domain is described with a polytope as we have discussed in Section 2.1. For process $P1$, for example, the process iteration domain is defined as $D_{P1} = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 99 \wedge 2i \leq j \leq -2i + 396\}$. This corresponds to the control part, i.e., the two for-loops, of process $P1$ as can be seen in Figure 2.4.

Definition 7 *The **process iteration domain size**, denoted by $|D_{P_i}|$, represents the total number of iterations of process P_i.*

The process iteration domain size is obtained by counting the number of integer points in a polytope, which is supported by the polyhedral library PolyLib [98] as also indicated in Section 2.2. Calculating a process iteration domain sizes, is obviously the first prerequisite to estimate the total execution time of a process. It is used to evaluate the process splitting transformation and is further discussed in Chapter 3.

Finally, we give 3 definitions that are related to the communication in polyhedral process networks. First, we consider input port domains that implement the control to read data from FIFO channels, then we define a mapping function that specifies an iteration point where data is produced, and finally we define an output port domain that specifies a set of points that generate data for a particular input port.

Definition 8 *The n-th **input port domain** of process P_i, denoted by $IP^n_{P_i}$, is defined as a subset of the process iteration domain where data is read from the n-th incoming FIFO channel, i.e., $IP^n_{P_i} \subseteq D_{P_i}$.*

Consider process $P1$ in Figure 2.4, which has one input port domain IP^1_{P1}. This input port domain is read at each process iteration, which means that the input port domain contains the same points as the process iteration domain, i.e., $IP^1_{P1} = D_{P1} = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 99 \wedge 2i \leq j \leq -2i + 396\}$. But when we look at process $P2$, for example, then we see that the input data is sometimes read from IP^1_{P2} and in other cases from IP^2_{P2}, i.e. they are a subset of the process iteration domain. To be more specific, we see that $IP^1_{P2} = \{(i,j) \in \mathbb{Z}^2 \mid i + j - 299 \geq 0\} \cap D_{P_2}$, and that $IP^2_{P2} = \{(i,j) \in \mathbb{Z}^2 \mid -i + j + 298 \geq 0\} \cap D_{P_2}$, where $D_{P_2} = \{(i,j) \in \mathbb{Z}^2 \mid 49 \leq i \leq 149 \ \wedge \ i + 101 \leq j \leq 250\}$. Note that these two input ports are mutually exclusive. The reason is that the `write_data` process function has only one input argument and process $P2$ needs only one input token per process iteration from one of these two input ports.

In a similar way, we define an output port domain which represents the process iterations for which writing data to a particular FIFO channel occurs.

Definition 9 *The n-th **output port domain** of a process P_i, denote by $OP^n_{P_k}$, is defined as the subset where data is written to the n-th outgoing FIFO channel, i.e., $OP^n_{P_k} \subseteq D_{P_k}$.*

When we first consider process $P1$ from Figure 2.4 again, we see that it has one output port, which is active at each process iteration, i.e., $OP^1_{P1} = D_{P1}$. A more complicated example is the first output port domain OP^1_{P0} of process $P0$. It can be seen that it is active only at particular iterations, i.e., $OP^1_{PO} = \{(i,j) \in \mathbb{Z}^2 \mid j/4 - i/2 \geq 0 \wedge -j/4 - i/2 + 99 \geq 0\} \cap D_{P_0}$, where $D_{P_0} = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i \leq 200 \ \wedge \ 0 \leq j \leq 400\}$. The reason that divisions of j and i by 4 and 2 appear in the constraints, is the result of consumer process $P1$ reading data in a particular pattern, i.e., array a is accessed with $2 * i$ and $4 * j$, see `line 8` in Figure 2.2.

We have defined input and output port domains that specify at which process iterations data is read/written. Thus, producer/consumer pairs of processes are connected via the output port domain of the producer with the input port domain of the consumer. Besides the port domains that specify when data is read/written, there is mapping function that specifies the relation between the input and output port domains, i.e., we define an affine mapping function M which maps the consumer process iterations to the producer iterations where the data is produced:

Definition 10 *We define an affine **mapping function** M^k as a function that maps the process iteration points from the k-th input port domain of a consumer process P_i to the process iteration points of the corresponding producer process P_j, i.e., $OP^l_{P_j} = M^k(IP^k_{P_i})$.*

An example is given in Figure 2.3. Let us consider process iteration $(1,3)$ of the

consumer process that executes function `filter` as shown in Figure 2.3 B). Since data is read from $a[2 * i][4 * j]$, process iteration $(1, 3)$ reads data that is produced at iteration $(2 * 1,\ 4 * 3) = (2, 12)$. This point is marked in the producer iteration domain as shown in Figure 2.3 A). Thus, we have a mapping function $M : \begin{pmatrix} i_p \\ j_p \end{pmatrix} = [2\ 4] \begin{pmatrix} i_c \\ j_c \end{pmatrix}$, where $\begin{pmatrix} i_p \\ j_p \end{pmatrix} \in OP^1_{P_0}$ and $\begin{pmatrix} i_c \\ j_c \end{pmatrix} \in IP^1_{P_1}$. This mapping function is also shown in Figure 2.3, and maps consumer iteration $(i_c = 1, j_c = 3)$ to its corresponding producer iteration $(i_p = 2, j_p = 12)$.

2.6 Validity of Transformations

In this section, we briefly review the validity of the process transformations presented in Chapters 3, 4, and 5. That is, we indicate that we can always apply the process splitting and merging transformations in a valid way for any given PPN. We first look at the validity of statement reordering transformations for sequential input programs, because the same constraints apply for process transformations in PPNs. Therefore, we discuss the different types of data dependencies that can exist between program statements that should be respected when the program code is transformed, which ensures that the transformed program code is input/output equivalent with the original program code.

As data dependence analysis is such a crucial step in program transformations, it is extensively researched and discussed in the literature [2, 6, 43, 57]. The goal of data dependence analysis is to find dependent program statements that read/write data from/to the same memory location. Three different data dependence relations can be identified. A flow or *true* dependence exist between two program statements A and B when A produces data that is read by B. This is denoted by $A\ \delta^f\ B$. The two other dependence relations are *anti* and *output* dependencies. In case of an anti-dependence, data is first read by a statement A and then written by statement B, which is denoted by $A\ \delta^a\ B$. An output-dependence exists when two statements A and B write to the same memory location, which is denoted by $A\ \delta^o\ B$. These data dependencies can also exists between different executions of statements within a loop-nest. If that is the case, then we say that the dependence is *loop-carried*.

As already mentioned, the data dependence information is used in optimizations and transformations to ensure correct behavior of the transformed program code. Transforming the program code is valid as long as the data dependencies are not changed. In our analysis to derive PPNs from static affine nested loop programs, we use *exact* array data flow analysis [25, 71]. This means that dependencies between statements are represented by exact dependency relations in the form of an affine combination of loop iterators and program parameters. Thus, the dependencies are

not abstracted with, for example, direction or distance vectors. An example of an exact dependence relation is the mapping function shown in Figure 2.3. It maps an iteration point of the consumer iteration space to an iteration point of the consumer where the data is produced. Taking into account these exact data dependence relations between consumer and producer statements, makes it possible to apply the process splitting and merging transformations in a valid way for any given PPN. That is, for the splitting transformation, more processes are introduced and the dependencies are recalculated to ensure that the processes communicate data in the proper way. For the process merging transformation, the executions of different processes are merged into a single process. The polyhedra that describe the process iterations of different processes, are merged using the work and code generator described in [7]. The merging is done pairwise for two given processes and the validity is checked using the exact dependence relations as described in [92, 94].