Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

Chapter 2

Background

In order to comprehend the next chapters, this chapter contains some basic material from the theory of integer linear algebra. Besides introduction of notations and definitions, this chapter deals with models of computation and compiler techniques used for parallelizing sequential programs.

Further, this chapter is organized as follows. Section 2.1 gives some notations and definitions used throughout the dissertation. We present the Polyhedral Model and show how this model can be extracted from SANLPs. Section 2.2 presents the formal definitions of the program models of dynamic applications introduced in Chapter 1. The parallelization approach presented in this dissertation deals with this type of dynamic programs only. Section 2.3 presents the definition of Polyhedral Process Networks model of computation which is used as a target parallel model of computation.

For better understanding of the solution approaches presented in the following Chapters, we give a brief overview of the two state-of-the-art techniques used to analyze sequential programs. The first one, called Exact Array Dataflow Analysis (EADA) [4], is used to analyze static programs, namely SANLPs. Recall, that EADA is implemented in the *pn* [48] compiler for the translation of SANLPs to PPN. We present EADA in Section 2.4.

The second technique, which we present in Section 2.5, allows for the analysis of programs with more relaxed constraints than SANLPs. That is, we consider the Fuzzy Array Dataflow Analysis (FADA) introduced in [37, 38]. FADA is an enhanced version of EADA and it is used to analyze programs with dynamic behavior.

Finally, Section 2.6 briefly presents important definitions and theory used to identify communication models while deriving a Polyhedral Process Network specification.

2.1 Preliminaries

The formal objects handled in this dissertation are mainly vectors with integer coordinates. A sub-vector of a vector $\vec{x}$ built from components k to l is written as: $\vec{x}[k..l]$. Similarly, $\vec{x}[i]$ is a shorthand for $\vec{x}[i..i]$. By $\ll$ we denote lexicographical ordering of vectors. This is expressed as a set of equalities and inequalities as:

$$\vec{a} \ll \vec{b} \equiv \bigvee_{i=1}^{n} \left(\vec{a}[i] < \vec{b}[i] \wedge \bigwedge_{j=1}^{i-1} \vec{a}[j] = \vec{b}[j] \right) \tag{2.1}$$

The smallest and the largest vectors according to $\ll$ are the lexicographical minimum ($lexmin$) and lexicographical maximum ($lexmax$), respectively.

2.1.1 Polyhedral Model

Sets of rational values described by affine inequalities have been the subject of extensive research and are called polyhedra.

Definition 2.1.1 (polyhedron)
The implicit definition of polyhedron is defined as the intersection of a finite set of closed linear half-spaces. Polyhedron is specified by a system of linear inequalities and equalities:

$$\mathcal{P} : \{\vec{x} \in \mathbb{Q}^n | A\vec{x} \geq \vec{b}\} \tag{2.2}$$

where A is a $j \times n$ matrix, $\vec{b}$ is a j-vector, and where n is the dimension of the space containing the polyhedron. The dimension of a polyhedron is defined to be the dimension of the smallest affine subspace which spans the polyhedron. A polyhedron of dimension d is called a d-polyhedron. Z-polyhedron [6] denotes a polyhedron whose points are integers.

Definition 2.1.2 (parameterized polyhedron)
The parameterized polyhedra is a family of polyhedra $\mathcal{P}(\vec{p})$ described as a linear function of $\vec{p}$, which is an m-vector of parameters:

$$\mathcal{P}(\vec{p}) = \{\vec{x} \in \mathbb{Q}^n | A\vec{x} + B\vec{p} \geq \vec{c}\}, \vec{p} \in \mathbb{Q}^m \tag{2.3}$$

where A and B are constant matrices and $\vec{c}$ is a constant vector.

In the compilers domain, the input program is usually represented in some internal representation form. This form allows for manipulation and optimization, for example in the context of loop transformations. One of this special intermediate program formats called Polyhedral Model was originally introduced for systolic array synthesis but also was found useful for parallelizing compilers [4]. This model applies to the class of affine nested loop programs and is used in compiler optimizations to efficiently analyze and transform the input program.

In the following, we demonstrate how the Polyhedral Model can be extracted from sequential programs considered in this dissertation.

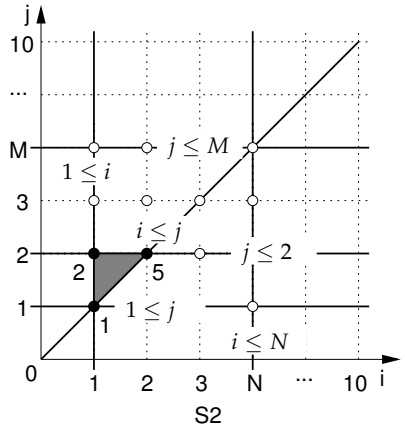


Figure 2.1: Geometrical representation of iteration domain of statement S2 in the program depicted in Figure 1.2(a).

The whole execution of a statement in a program can be described by the following constructs:

Iteration domain

The Iteration Domain (ID) is the set of values of an iteration vector for which a statement is executed. ID of a statement S is denoted by $\mathbf{D}(S)$. An *iteration vector* $\vec{x}$ of a statement in a dynamic program is built from iterators of surrounding for- and while-loops. Although, an iterator for a *while*-loop may not be explicitly mentioned in the source code of a program, we can associate some "virtual" iterator $w : 0 \leq w$ with the *while*-loop.

Because the execution of a statement is guarded by an affine control, its iteration domain can be specified as a set of linear inequalities defining a Z-polyhedron. For example, consider statement $S2$ in Figure 1.2(a). Its iteration domain represented in algebraic form is the following parameterized polyhedron:

$$\begin{aligned}\mathbf{D}(\mathrm{S2}) = \mathcal{P}(M,N) \quad &= \\ &= \left\{ (i,j) \in \mathbb{Q}^2 \middle| \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{bmatrix} 1 \\ -N \\ 0 \\ -M \\ -2 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} M \\ N \end{pmatrix} \geq \begin{bmatrix} 1 \\ -10 \\ 1 \\ -10 \end{bmatrix} \right\} = \\ &= \{(i,j) \in \mathbb{Q}^2 | 1 \leq i \leq N \wedge i \leq j \leq M \wedge j \leq 2 \wedge 1 \leq M \leq 10 \wedge 1 \leq N \leq 10\}.\end{aligned}$$

For illustrative purposes, the ID of statement S2 in a graphical form is shown in Figure 2.1. Similarly, the iteration domain in algebraic form of statement $S2$ shown in Figure 1.5(c) is:

$$\mathbf{D}(S2) = \{(i, w) \in \mathbb{Q}^2 \mid 1 \leq i \leq N \wedge 0 \leq w \wedge 1 \leq N \leq 10\}.$$

Order of execution

In an affine nested loop programs statements evaluate some data. An evaluation of a single statement W on iteration point $\vec{x}$ is called an *operation* and is denoted as $\langle W, \vec{x} \rangle$, where $\vec{x} \in \mathbf{D}(\mathrm{W})$.

The schedule determines the execution order of all operations of all statements in a program. The execution order of operations can be established using the sequencing predicate $\prec$. An operation $\langle W, \vec{x} \rangle$ is evaluated before an operation $\langle R, \vec{y} \rangle$ ($\langle W, \vec{x} \rangle \prec \langle R, \vec{y} \rangle$) according to the program sequence if: 1) iteration point $\vec{x}$ lexicographically precedes iteration point $\vec{y}$; or 2) if $\vec{x} = \vec{y}$ and statement W precedes statement R in the program code. The sequencing predicate depends only on the code of a sequential program. Let N_{WR} be the number of loops enclosing both statement W and R. Let $\lhd$ be the textual order of statements W and R in the code of the program. Then the execution order is given by:

$$\langle W, \vec{x} \rangle \prec \langle R, \vec{y} \rangle \equiv \vec{x}[1..N_{WR}] \ll \vec{y}[1..N_{WR}] \vee (\vec{x}[1..N_{WR}] = \vec{y}[1..N_{WR}] \wedge W \lhd R) \quad (2.4)$$

[4] shows how sequencing predicate $\prec$ can be expanded to a system of linear inequalities.

2.2 The Program Model

In the following, we will give definitions of the type of sequential programs we consider in this dissertation.

Definition 2.2.1 (static affine nested loop program,SANLP)
A **static affine nested loop program (SANLP)** is a program where each program statement is enclosed by one or more for-loops and `if`-statements, and where:

1. loops have a constant step size;
2. loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;
3. `if`-statements have affine conditions in terms of the loop iterators, static program parameters, and constants;
4. index expressions of array references are affine functions of the enclosing loop iterators, static program parameters, and constants;

5. data flow between statements is explicit via a variable or an array.

An example of a SANLP is given in Figure 1.2(a).

Definition 2.2.2 (Weakly Dynamic Program,WDP)
A **Weakly Dynamic Program (WDP)** is a program where each program statement is enclosed by one or more for-loops and `if`-statements, and where:

1. loops have a constant step size;
2. loops have bounds that are affine expressions of the enclosing loop iterators, static program parameters, and constants;
3. `if`**-statements have no restrictions on conditions** - the condition of *if* can be an arbitrary function of program variables, enclosing loop iterators, static program parameters, and constants;
4. index expressions of array references are affine functions of the enclosing loop iterators, static program parameters, and constants;
5. data flow between statements is explicit via a variable or an array.

An example of a WDP program is given in Figure 1.5(a).

Definition 2.2.3 (affine nested loop program with dynamic loop bounds, Dynloop)
An **affine nested loop program with dynamic loop bounds (Dynloop)** is a program where each program statement is enclosed by one or more for-loops and `if`-statements, and where:

1. loops have a constant step size;
2. **loops have no restrictions on the bounds** - the bounds of for-loops can be an arbitrary expression of program variables, the enclosing loop iterators, static program parameters, and constants;
3. `if`**-statements have no restrictions on conditions** - the condition of `if` can be an arbitrary function of program variables, enclosing loop iterators, static program parameters, and constants;
4. index expressions of array references are affine functions of the enclosing loop iterators, static program parameters, and constants;
5. data flow between statements is explicit via a variable or an array.

An example of a Dynloop program is given in Figure 1.5(b).

Definition 2.2.4 (affine nested loop program with while-loops,WLAP)
An **affine nested loop program with while-loops (WLAP)** is a program where each program statement is enclosed by one or more for-loops, **while-loops** and `if`-statements, and where:

1. for-loops have a constant step size;
2. loops have no restrictions on the bounds - the bounds of for-loops can be an arbitrary expression of program variables, the enclosing loop iterators, static program parameters, and constants;
3. `if`-statements have no restrictions on conditions - the condition of *if* can be an arbitrary function of program variables, enclosing loop iterators, static program parameters, and constants;
4. index expressions of array references are affine functions of the enclosing loop iterators, static program parameters, and constants;
5. data flow between statements is explicit via a variable or an array.

An example of a WLAP program is given in Figure 1.5(c).

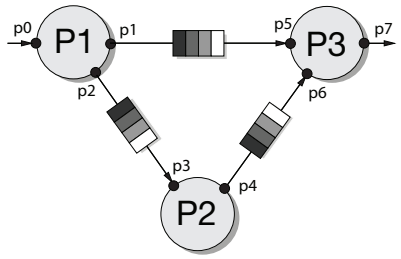
2.3 Polyhedral Process Networks

Below, we give a definition of the Polyhedral Process Network Model of Computation.

Definition 2.3.1 (Polyhedral Process Network,PPN)
The PPN model of computation is a special case of the Kahn Process Networks (KPN) [7] model of computation with the following properties:

- it consists of concurrent autonomous processes;
- processes communicate data in a point-to-point fashion over bounded FIFO channels via ports;
- processes synchronize via blocking read/write on an empty/full FIFO;
- processes have a well defined structure consisting of read, execute and write code sections;
- it is deterministic;
- it has a distributed control.

An example of a PPN is illustrated in Figure 2.2(a). The PPN consists of three processes, P1, P2 and P3, and three FIFO channels. The examples of code of processes P1 and P3 of this PPN are illustrated in Figure 2.2(b) and Figure 2.2(c), respectively. In order to see the well defined structure of every process on a PPN, consider the source code of process P3 in Figure 2.2(c). In the *read* section at lines 3–7, the process reads data from two ports $p5$ or $p6$. In the *execute* section at line 8, the process executes function F3() on data that has been read. In the *write* section at line 9, the process



(a) An example of a PPN.

```
1  int M = 10, P = 3

2  for i = 1 to M,
3    out = F1()
4    if i <= P,
5      write(p1, out)
6    else
7      write(p2, out)
8    endif
9  endfor
```

(b) An example of process P1.

```
1  int N = 10, P = 3

2  for j = 1 to N,
3    if j <= P,
4      in = read(p5)
5    else
6      in = read(p6)
7    endif
8    out = F2(in)
9    write(p7, out)
10 endfor
```

(c) An example of process P3.

Figure 2.2: An example of a PPN and source codes of its processes P1 and P3.

writes the produced data to port $p7$. This clearly separated structure of a process in a PPN allows for explicit separation between computation and communication.

In a PPN, every process can be described by the following terms.

Definition 2.3.2 (node domain)
A Node Domain (ND) of process P executing function F in a PPN is the set of iteration points ND_{P_F} for which function F is executed.

In this dissertation we consider PPNs where every process executes only one function, and, therefore, for the sake of brevity, we can use $ND_{P_F} = ND_P$. A node domain of a process can be represented by a polyhedron. In Section 2.1.1, it has been demonstrated that a set of iteration points can be represented as a Z-polyhedron. Similarly, a node domain of a process can be represented as a Z-polyhedron. For example, consider process P1 of PPN shown in Figure 2.2(a). The code of process P1 is illustrated in Figure 2.2(b). The process executes function $F1()$, and, thus, its node domain is $\mathbf{D}(ND_{P1}) = \{i \in \mathbb{Z} | 1 \leq i \leq M \wedge M = 10\}$.

Definition 2.3.3 (input port domain)
An input port domain (IPD) of port p is the set of iteration points $I_p \in IPD_p$ for which port p is read.

Definition 2.3.4 (output port domain)
An output port domain (OPD) of port q is the set of iteration points $O_q \in OPD_q$ for which port q is written.

Similarly to a node domain, IPDs and OPDs of every process can be represented as Z-polyhedra. For example, consider source codes of processes P1 and P3 depicted in Figure 2.2(b) and Figure 2.2(c), respectively. Processes are connected via output port $p2$ (see line 7 in Figure 2.2(b)) of process P1 and input port $p5$ (see line 4 in Figure 2.2(c)) of process P3. Therefore, input and output port domains of ports $p5$ and $p2$ are $\mathbf{D}(IPD_{p5}) = \{j \in \mathbb{Z} | 1 \leq j \leq N \wedge j \leq P \wedge N = 10 \wedge P = 3\}$, and $\mathbf{D}(OPD_{p2}) = \{i \in \mathbb{Z} | 1 \leq i \leq M \wedge i > P \wedge M = 10 \wedge P = 3\}$.

A FIFO in a PPN is connected to processes which writes and reads via ports. For every FIFO, there exists a mapping function that maps the iteration points of IPD of the process that reads from the FIFO to the iteration points of OPD of the process that writes to the FIFO. Consider a FIFO connected to processes via ports p and q.

Definition 2.3.5 (mapping function)
A mapping function is an affine mapping $f_{pq} : I_p \rightarrow O_q : O_q = f(I_p)$, where $I_p \in IPD_p$ and $O_q \in OPD_q$.

An example of the mapping function between ports p_1 and p_5 of processes $P1$ and $P3$ in the PPN shown in Figure 2.2(a) is $f_{p_5p_1} : \mathbb{Z} \rightarrow \mathbb{Z} : j = i * 1, i \in IPD_{p_5}, j \in OPD_{p_1}$.

2.4 Exact Array Dataflow Analysis

Because our approach of parallelizing dynamic programs presented in the following chapters is an extension of the parallelization approach of static programs, for better understanding, in this section we formally describe the EADA [4] algorithm, which is used to perform dependence analysis on static programs only. We will demonstrate an application of the EADA algorithm on the static program depicted in Figure 1.2(a).

The goal of the dependence analysis is to determine if evaluation of a statement depends on evaluation of other statements and to find these evaluations. For example, in the SANLP program depicted in Figure 1.2(a), the purpose of the dependence analysis is to find whether statement $S3$ depends on statements $S1$ or $S2$ via array y and at which iterations. Or in other words, for every element of array y read at a given iteration of statement S3, the dependence analysis finds which statement, $S1$ or $S2$, and at which iteration it writes data to the given array element. The result of the analysis forms the dependency relations between iterations of statements writing/reading to/from the array.

Consider two statements W and R, and operations $\langle W, \vec{x} \rangle$ and $\langle R, \vec{y} \rangle$, where the first operation writes to an array and the second operation reads from it. The operation $\langle W, \vec{x} \rangle$ is a source for operation $\langle R, \vec{y} \rangle$ if it satifsies the system of linear (in)equalities (2.5). Note, that all iteration vectors of operations that satisfy this system form a convex domain.

$$\begin{aligned} \mathbf{Q}_{WR}(\vec{y}) = \{\vec{x} \mid\ & \vec{x} \in \mathbf{D}(W), && \text{(c1)} \\ & \mathcal{I}_W(\vec{x}) = \mathcal{I}_R(\vec{y}), && \text{(c2)} \\ & \langle W, \vec{x} \rangle \prec \langle R, \vec{y} \rangle\}. && \text{(c3)} \end{aligned} \tag{2.5}$$

The first constraint (c1) states that the source iteration $\vec{x}$ has to exist, i.e., it has to belong to the iteration domain of statement W. The constraint (c2) specifies that if

there is a dependency between two operations, both have to access the same array element. To access an array element, operation $\langle W, \vec{x} \rangle$ uses an affine indexing function $\mathcal{I}_W()$ and operation $\langle R, \vec{y} \rangle$ uses an affine indexing function $\mathcal{I}_R()$. The (c3) constraint determines an order of operations, i.e., source operation $\langle W, \vec{x} \rangle$ has to be evaluated *before* operation $\langle R, \vec{y} \rangle$.

There might be many operations of a single statement satisfying system (2.5), i.e., writing to the same array element. However, only the "last" writing operation is the source for operation $\langle R, \vec{y} \rangle$. Therefore, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(\vec{y})$:

$$\mathbf{K}_{WR}(\vec{y}) = lexmax\{\mathbf{Q}_{WR}(\vec{y})\}. \tag{2.6}$$

So far, operations of only single statement have been considered, while there might be several statements $W_1,\ldots,W_m$ writing to the same array element. In this case, all pairs $W_1/R,\ldots W_m/R$ have to be considered. The actual source is the "last" operation between all operations of all statements:

$$\sigma(\langle R, \vec{y} \rangle) = lexmax\{\langle W_k, \mathbf{K}_{W_kR}(\vec{y}) \rangle \mid k \in [1, m]\}. \tag{2.7}$$

For example, consider the program in Figure 1.2(a). There are two statements, S1 and S2 writing to array y and one statement $S3$ reading from that array. Therefore, we consider two pairs S1S3 and S2S3. For each pair we build the system of linear inequalities (2.5) as depicted in Table 2.1 (see $\mathbf{Q}_{\text{S1S3}}((i_3, j_3))$ and $\mathbf{Q}_{\text{S2S3}}((i_3, j_3))$). With (i_3, j_3), we denote the iteration vector (i, j) of statement $S3$.

$\mathbf{Q}_{\text{S1S3}}((i_3, j_3))$	$\mathbf{Q}_{\text{S2S3}}((i_3, j_3))$	
$1 \leq k \leq M$	$1 \leq i_2 \leq N \wedge i_2 \leq j_2 \leq M \wedge$ $j_2 \leq 2$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2) \rangle \prec \langle S3, (i_3, j_3) \rangle$	(c3)

Table 2.1: Examples of system (2.5) for S1S3 and S2S3 statements.

Finding lexicographical maximums, $\mathbf{K}_{\text{S1S3}}()$ and $\mathbf{K}_{\text{S2S3}}()$, of the systems in Table 2.1 means to solve the Parametric Integer Linear Problems (PILPs) depicted in Table 2.2. The solution to find the maximum point for a given convex domain is based on the dual simplex method [16] that is implemented in open-source libraries such as *isl* [17], Parma Polyhedral Library [18], and PIPLib [19].

The source operation $\sigma(\langle \text{S3}, () \rangle)$ is found by determining the *lexmax* between $\mathbf{K}_{\text{S1S3}}()$ and $\mathbf{K}_{\text{S2S3}}()$ which is another PILP problem. Finally, the source operation $\sigma(\langle \text{S3}, (i_3, j_3) \rangle)$ for the data read by statement S3 can be written in the following form:

Objective:	$lexmax\{(i_3, j_3)\}$	$lexmax\{(i_3, j_3)\}$
subject to:	$\mathbf{Q}_{S1S3}((i_3, j_3))$	$\mathbf{Q}_{S2S3}((i_3, j_3))$

Table 2.2: PILP problems for pairs S1S3 and S2S3.

$$\sigma(\langle S3, (i_3, j_3)\rangle) = \left| \begin{array}{ll} \textbf{if} & j_3 \leq 2 \\ \textbf{then} & \langle S2, (i_3, j_3)\rangle \\ \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \right. \tag{2.8}$$

Both branches of the `if`-statement in Solution (2.8) shown above represent solutions of the PILP problems formulated in Table 2.2. The `if`-condition is derived by finding the lexicographical maximum by Equation 2.7. Solution (2.8) can be interpreted as follows: the source of the data for statement $S3$ of the program in Figure 1.2(a) can be two statements – the source is statement $S1$ when the iterator j of $S3$ is greater than 2, otherwise, the source is statement $S2$.

2.5 Fuzzy Array Dataflow Analysis

As explained in Section 1.3, it is impossible to apply the EADA dependence analysis algorithm to dynamic programs. However, there exists an enhanced version of the EADA algorithm called Fuzzy Array Dataflow Analysis (FADA) [37, 38]. FADA allows for the compile-time dependence analysis of programs where arbitrary `if`-conditions and *while*-loops are allowed. We formally describe FADA because it is an important part of our parallelization approaches presented in the following chapters.

In order to simplify the explanation of the FADA algorithm, we split our presentation in 2 parts. In the first part, we formally present the application of the FADA analysis on programs containing dynamic `if`-conditions only. In the second part, we present an application of the FADA algorithm on programs containing *while*-loops only. In general, the FADA algorithm combines both methods.

I. dynamic `if`-conditions

Consider two statement W and R of a dynamic program. Operation $\langle W, \vec{x}\rangle$ writes to and operation $\langle R, \vec{y}\rangle$ reads from the same array. Moreover, let statement W be surrounded by a data-dependent `if`-condition. As a running example, consider Figure 1.5(a): statements $S2$ and $S3$ are W and R, respectively, and the `if`-condition at line C surrounding statement $S2$ is a data-dependent condition.

In Section 2.4, it has been shown that in order to have two operations $\langle W, \vec{x}\rangle$ and $\langle R, \vec{y}\rangle$ of a *static* program dependent, they have to comply to the system of linear

inequalities (2.5). In the same way, to find whether operation $\langle W, \vec{x} \rangle$ is a source for operation $\langle R, \vec{y} \rangle$ in a *dynamic* program, the following system of linear inequalities is built:

$$\begin{aligned} \mathbf{Q}_{WR}(\vec{y}, \vec{\alpha}) = \{\vec{x} \mid\ & \vec{x} \in \mathbf{D}(W), \vec{x} = \vec{\alpha}, && (c1) \\ & \mathcal{I}_W(\vec{x}) = \mathcal{I}_R(\vec{y}), && (c2) \\ & \langle W, \vec{x} \rangle \prec \langle R, \vec{y} \rangle \}. && (c3) \end{aligned} \tag{2.9}$$

The meaning of constraints $(c2)$ and $(c3)$ is the same as in system (2.5): operations should access the same array element and the writing operation should occur before the reading operation. We will explain the meaning of constraint $(c1)$. As statement W is surrounded by data-dependent `if`-condition, exact operations of W cannot be determined at compile-time. Thus, for any reading operation $\langle R, \vec{y} \rangle$ it is impossible to determine the *exact* source operation. The idea of the FADA algorithm is to introduce a parameter which would hide unknown information, i.e., a parameter is used to indicate at which iteration a writing operation $\langle W, \vec{x} \rangle$ may occur. It is unknown exactly at which iteration points $\vec{x} \in \mathbf{D}(W)$ writing to the array occurs, but it is assumed that this happens for iterations $\vec{x} = \vec{\alpha}$, where $\vec{\alpha}$ is a free parameter vector whose values have to be determined at run-time. Because source operations satisfying system (2.9) are not exact, we call them *approximated* sources.

Similarly to the EADA algorithm, only the "last" writing operation is the source for $\langle R, \vec{y} \rangle$. Therefore, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(\vec{y}, \vec{\alpha})$:

$$\mathbf{K}_{WR}(\vec{y}, \vec{\alpha}) = lexmax\{\mathbf{Q}_{WR}(\vec{y}, \vec{\alpha})\}. \tag{2.10}$$

Finally, the FADA algorithm considers all statements $W_1, \ldots, W_m$ which write to the same array element. For each W_k, $k \in [1..m]$, the approximated sources (2.10) are found. Finally, the source operation is found by combining all approximated sources as shown in (2.11). The procedure covering in depth the combination of all approximated sources is described in-depth in [37, 38].

$$\sigma(\langle R, \vec{y} \rangle, \vec{\alpha}) = lexmax\{\langle W_k, \mathbf{K}_{W_k R}(\vec{y}) \rangle | \quad k \in [1, m]\}. \tag{2.11}$$

For example, consider the program depicted in Figure 1.5(a). There are two statements $S1$ and $S2$ writing to array `y[]` and one statement $S3$ which reads from it. For every pair $S1S3$ and $S2S3$, the systems of linear inequalities (2.9) are built which are depicted in Table 2.3. For pair $S1S3$ all operations of statement $S1$ are known and thus, a parameter is not introduced (see system $\mathbf{Q}_{S1S3}((i_3, j_3))$ in Table 2.3). However, for pair $S2S3$ (see system $\mathbf{Q}_{S2S3}((i_3, j_3), (\alpha_i, \alpha_j))$), the parameter vector $\vec{\alpha} = (\alpha_i, \alpha_j)$ is introduced. This parameter vector is needed as statement $S2$ is surrounded by the dynamic `if`-condition at line C in Figure 1.5(a) and, thus, exact operations of $S2$ cannot be determined at compile-time. These parameters are used to

designate at which iteration of $S2$ a writing to the array `y[]` may occur. Values of the parameters are determined at run-time.

$\mathbf{Q}_{S1S3}((i_3, j_3))$	$\mathbf{Q}_{S2S3}((i_3, j_3), (\alpha_i, \alpha_j))$	
$1 \leq k \leq M$	$1 \leq i_2 \leq N \wedge i_2 \leq j_2 \leq M \wedge$ $i_2 = \alpha_i \wedge j_2 = \alpha_j$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

Table 2.3: Examples of system (2.9) for S1S3 and S2S3 statements.

Approximated sources $\mathbf{K}_{S1S3}()$ and $\mathbf{K}_{S2S3}()$ are found by solving the parametric integer linear problems (PILPs), similar to the ones presented in Table 2.2. Finally, the source operation defined in Equation (2.11) is determined by the recurrent algorithm of combining direct dependencies described in Section 5.2 of [37]. Therefore, the source operation for statement S3 is:

$$\sigma(\langle S3, (i_3, j_3)\rangle, (\alpha_i, \alpha_j)) = \left| \begin{array}{l} \textbf{if} \quad i_3 \geq \alpha_i \wedge j_3 = \alpha_j \\ \textbf{then} \quad \langle S2, (\alpha_i, \alpha_j)\rangle \\ \textbf{else} \quad \langle S1, (j_3)\rangle. \end{array} \right. \tag{2.12}$$

From Solution (2.12) above, it can be seen that for any read operation $\langle S3, (i_3, j_3)\rangle$ there are two data sources: statements $S1$ or $S2$. When for a given iteration (i_3, j_3) of statement $S3$, at least one of the previous evaluations of the condition at line C in Figure 1.5(a) was `true`, then parameter $\alpha_i \leq i_3$ and, parameter $\alpha_j = j_3$, thus, the source is statement $S2$. Otherwise, the source is statement $S1$. In contrast to Solution (2.8), Solution (2.12) is approximated, because it depends on parameters (α_i, α_j) that are determined at run-time.

II. while loops

Consider again two statements W and R of a dynamic program. Operation $\langle W, \vec{x}\rangle$ writes to and operation $\langle R, \vec{y}\rangle$ reads from the same array. Moreover, statement W is enclosed in a *while*-loop at depth d. As a running example, consider Figure 1.5(c): statements $S2$ and $S3$ are W and R, respectively; statement $S2$ is enclosed in the while-loop at depth 1. The iteration vector of statement $S2$ is $\vec{x} = (i, w)$. To find whether operation $\langle W, \vec{x}\rangle$ is a source for operation $\langle R, \vec{y}\rangle$, the following system of linear inequalities is built:

$$\begin{aligned}\mathbf{Q}_{WR}(\vec{y},(\vec{\alpha},\beta)) = \{\vec{x} \mid\ & \vec{x} \in \mathbf{D}(W), \vec{x}[1..d] = \vec{\alpha}, \\ & 1 \le \vec{x}[d+1] \le \beta && (c1) \\ & \mathcal{I}_W(\vec{x}) = \mathcal{I}_R(\vec{y}), && (c2) \\ & \langle W,\vec{x}\rangle \prec \langle R,\vec{y}\rangle\}. && (c3)\end{aligned} \tag{2.13}$$

The meaning of constraints $(c2)$ and $(c3)$ is the same as in system (2.5): operations should access the same array element and the writing operation should occur before the reading operation. We will explain the meaning of constraint $(c1)$. As statement W is surrounded by a while-loop, exact operations of W cannot be determined at compile-time. Thus, for any reading operation $\langle R,\vec{y}\rangle$ it is impossible to determine the *exact* source operation. The idea of the FADA algorithm is to introduce parameters which would hide unknown information, i.e., parameters are used to indicate at which iteration a writing operation $\langle W,\vec{x}\rangle$ may occur. We do not know exactly at which iteration $\vec{x} \in \mathbf{D}(W)$ writing to the array occurs, but we assume that this happens for iterations $\vec{x}[1..d] = \vec{\alpha}$ and $1 \le \vec{x}[d+1] \le \beta$. Vector $\vec{x}[1..d]$ is built of iterators enclosing the while-loop, and iterator $\vec{x}[d+1]$ is the while-loop iterator. Parameter vector $\vec{\alpha}$ captures the values of loop iterators *enclosing* the while-loop, and parameter β indicates the upper bound of the while-loop, i.e., we introduce a parameter vector $(\vec{\alpha},\beta)$. Both parameters are free parameters which values have to be determined at run-time. Because source operations satisfying system (2.13) are not exact, we call them *approximated* sources.

Similar to systems (2.10) and (2.11) the following systems define the source operation.

$$\mathbf{K}_{WR}(\vec{y},(\vec{\alpha},\beta)) = lexmax\ \ \mathbf{Q}_{WR}(\vec{y},(\vec{a},\beta)). \tag{2.14}$$

$$\sigma(\langle R,\vec{y}\rangle,(\vec{\alpha},\beta)) = lexmax\{\langle W_k,\mathbf{K}_{W_kR}(\vec{y},(\vec{\alpha},\beta))\rangle \mid k \in [1,m]\}. \tag{2.15}$$

To illustrate this algorithm, consider the WLAP depicted in Figure 1.5(c). There are two statements $S1$ and $S2$ writing to array `y[]` and one statement $S3$ which reads from it. For every pair $S1S3$ and $S2S3$ the systems of linear inequalities (2.13) are built. The systems are depicted in Table 2.4. To capture all evaluations of statement $S2$, the new iterator w is introduced which corresponds to the while-loop at line 8. For pair $S1S3$ all operations of statement $S1$ are known and thus, a parameter is not introduced (see system $\mathbf{Q}_{S1S3}(i_3)$ in Table 2.4). However, for pair $S2S3$ (see system $\mathbf{Q}_{S2S3}(i_3,(\alpha,\beta))$ in Table 2.4), new parameters α and β are introduced as shown in system (2.13), because statement $S2$ is surrounded by the while-loop at line 8 in Figure 1.5(c) and, thus, exact operations of $S2$ cannot be determined at compile-time. These parameters are used to designate at which iteration of $S2$ a writing to the array `y[]` may occur. Values of the parameters are determined at run-time.

Approximated sources in $S1S3$ and $S2S3$ pairs are found by solving the parametric integer linear problems (PILPs) formulated similar to Table 2.2. Again, as in the

$\mathbf{Q}_{S1S3}(i_3)$	$\mathbf{Q}_{S2S3}(i_3,(\alpha,\beta))$	
$1 \leq i_1 \leq N$	$1 \leq i_2 \leq N \wedge$ $i_2 = \alpha \wedge 1 \leq w \leq \beta$	(c1)
$i_1 = i_3$	$i_2 = i_3$	(c2)
$\langle S1, i_1\rangle \prec \langle S3, i_3\rangle$	$\langle S2, (i_2, w)\rangle \prec \langle S3, i_3\rangle$	(c3)

Table 2.4: Examples of system (2.13) for $S1S3$ and $S2S3$ pairs.

previous section, the source operation defined in Equation 2.15 is determined by the recurrent algorithm of combining direct dependencies described in Section 5.2 of [37, 38]. Therefore, the source operation for statement S3 is:

$$\sigma(\langle S3, i_3\rangle, (\alpha,\beta)) = \left| \begin{array}{l} \textbf{if} \quad i_3 = \alpha \wedge \beta \geq 1 \\ \textbf{then} \quad \langle S2, (\alpha,\beta)\rangle \\ \textbf{else} \quad \langle S1, i_3\rangle. \end{array} \right. \tag{2.16}$$

From Solution 2.16 above, we see that for any read operation $\langle S3, i_3\rangle$ there are two data sources: statements $S1$ or $S2$. When for a given iteration i_3 of statement $S3$, there is an iteration of statement $S2$: $(i_2, w) = (\alpha,\beta)$, such that for $i_3 = \alpha$ there was at least one iteration of the while-loop, i.e., $\beta \geq 1$, then the source is statement $S2$. Otherwise, the source is statement $S1$. Solution 2.16 is approximated, because it depends on parameters (α,β) that are determined at run-time.

2.6 Communication model identification in PPNs derived from static programs

In this section, we consider important definitions and theory used in the Linearization step of the procedure of PPN derivation illustrated in Figure 1.2(b). In this step the multi-dimensional arrays are linearized and the communication models of all Producer/Consumer (P/C) pairs are identified. In order to understand our contribution presented in Chapter 5, we explain the communication model identification procedure on an example where a SANLP program is translated into a PPN.

Applying the EADA dependence analysis on the static program shown in Figure 1.2(a) allows to generate the PPN as shown in Figure 1.2(d). This PPN has three processes and two FIFO channels that connect processes via ports $p1$–$p4$. Thus, there are two P/C pairs: P1/P3 and P2/P3.

According to Definition 2.3.5, relations between reading/writing of processes in a P/C pair are expressed by the mapping functions. A mapping function in a P/C

pair gives for each iteration of a statement corresponding to a Consumer process, the iteration of a statement corresponding to a Producer process. For example, for the P1/P3 pair shown in Figure 1.2(d) connected via ports $p1$ and $p3$, the mapping function and its domain are:

$$f_{p_3p_1}: \quad \mathbb{Z}^2 \to \mathbb{Z} : k = (0\ 1)\begin{pmatrix} i_3 \\ j_3 \end{pmatrix}, \tag{2.17}$$
$$\mathbf{D}(f_{p_3p_1}) = \mathbf{D}(IPD_{p_3}) = \{(i_3, j_3) \in \mathbb{Z}\ \ | 1 \leq i_3 \leq j_3, 2 < j_3 \leq 4\},$$

and for P2/P3 pair connected via ports $p2$ and $p4$, the mapping function is:

$$f_{p_4p_2}: \quad \mathbb{Z}^2 \to \mathbb{Z}^2 : \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} i_3 \\ j_3 \end{pmatrix}, \tag{2.18}$$
$$\mathbf{D}(f_{p_4p_2}) = \mathbf{D}(IPD_{p_4}) = \{(i_3, j_3) \in \mathbb{Z} | 1 \leq i_3 \leq 2, i_3 \leq j_3 \leq 2\}.$$

The graphical representations of mapping functions (2.17) and (2.18) are illustrated in Figure 1.3(b). This figure depicts the iteration domains of statements $S1$, $S2$ and $S3$ which correspond to processes P1, P2 and P3 using the coordinate systems. The points on the coordinate systems designate the evaluations of statements and the arrows reflect the data dependency relations. The numbers at the points show the lexicographical order of statement evaluations. For pair P2/P3, mapping function $f_{p_2p_4}$ shown in Equation (2.18) maps points 1,2 and 5 from iteration domain of port $p4$ of statement S3 to points 1,2 and 5 of statement S1.

In Section 1.1.2, we explained that the communication model of a channel depends on the order of firings of the Producer and Consumer processes. We define the ordering in the communication models of a P/C pair as follows:

Definition 2.6.1 (in-order,out-of-order)
A P/C pair is **in-order** iff the mapping function f preserves the token order, i.e., every two Consumer iteration points $y^1, y^2 \in \text{LmP}(\mathbf{D}(f)) \wedge y^1 \ll y^2$ are mapped onto two Producer iteration points $x^1 = f(y^1)$ and $x^2 = f(y^2)$ such that $x^1 \leqq x^2$. If a P/C pair is not in order we call it **out-of-order**.

The LmP($\mathbf{D}(f)$) set used in Definition 2.6.1 is defined as follows:

Definition 2.6.2 (Lexicographically minimal Preimage,LmP)
Lexicographically minimal Preimage (LmP) is a set of the Consumer iteration points y_m that read the tokens from the Producer for the first time. LmP is found by solving the following Integer Linear Problem:

$$\begin{array}{ll} \textbf{objective}: & \textbf{subject to}: \\ y_m = lexmin\{f^{-1}(x)\}, & \begin{cases} y \in \mathbf{D}(f), \\ x = f(y). \end{cases} \end{array}$$

For example, in Figure 1.4(b), the LmP is marked by the dashed box and according to Definition 2.6.1 this P/C pair is **in-order**. Similarly, for our running example shown in Figure 1.3(b), the LmP corresponds to the dashed box and the communication model of a P/C pair formed by statement S2 and S3 is **in-order**.

The definition of multiplicity in a P/C pair given below we take from [14].

Definition 2.6.3 (multiplicity)
A P/C pair is **without multiplicity** iff the mapping function f is injective, i.e., $\forall\, y^1, y^2 \in \mathbf{D}(f))\ s.t.\ y^1 \neq y^2 \Rightarrow f(y^1) \neq f(y^2)$. Otherwise we say that the P/C pair is **with multiplicity**.

For example, in our running example shown in Figure 1.3(b), we see that there are at least two different iteration points of S3 which correspond to a single iteration point of S1. Therefore, the P/C pair formed by statements S1 and S3 has a **multiplicity**.

To analytically determine the communication type of an arbitrary P/C pairs in Figure 2.3 the *Reordering Problem* (RP) and the *Multiplicity Problem* (MP) are specified which correspond to Definitions 2.6.1 and 2.6.3, respectively.

$$\begin{cases} y^1, y^2 \in \mathrm{LmP}(\mathbf{D}(f)), \\ y^1 \ll y^2, \\ f(y^1) \gg f(y^2). \end{cases}$$

(a) Reordering Problem (RP)

$$\begin{cases} y^1, y^2 \in \mathbf{D}(f), & (c1) \\ y^1 \neq y^2, & (c2) \\ f(y^1) = f(y^2). & (c3) \end{cases}$$

(b) Multiplicity Problem (MP)

Figure 2.3: Reordering and Multiplicity Problems in static programs.

The RP and MP problems are integer linear problems (ILP), meaning that if, for example, there is an integer solution satisfying RP, then the communication model is *out-of-order*. Otherwise, the the communication model is *in-order*. Similarly, if there is an integer solution satisfying MP, then the communication model is *with-multiplicity*, and otherwise the communication model is *without-multiplicity*. For example, according to these problems, communication models of P/C pairs P1P3 and P2P3 in Figure 1.3(b) are IOM and IO, respectively.