Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks

Dmitry Nadezhkin

Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks.

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 20 december 2012
klokke 10:00 uur

door

Dmitry Nadezhkin
geboren te Arzamas-16, USSR
in 1981

Samenstelling promotiecommissie:

promotor	Prof.dr.ir. Ed F. Deprettere	Universiteit Leiden
co-promotor	Dr.ir. Todor Stefanov	Universiteit Leiden
overige leden:	Prof.dr. Albert Cohen	INRIA, Paris, France
	Prof.dr. Marco Bekooij	Universiteit van Twente
	Dr. Andy D. Pimentel	Universiteit van Amsterdam
	Prof.dr. Joost Kok	Universiteit Leiden
	Prof.dr. Harry Wijshoff	Universiteit Leiden

Parallelizing Dynamic Sequential Programs using Polyhedral Process Networks
Dmitry Nadezhkin. -
Thesis Universiteit Leiden. - With ref. - With summary in Dutch
ISBN 978-90-9027264-1

Printed in the Netherlands

To my parents Maria Alexandrovna and Alexander Trofimovich;
and my Masha

Contents

Chapter 1

Introduction

In the work entitled "Dialectics of Nature" Friedrich Engels (1883) formulated the law which can describe the universal law of nature, the material world and human society. According to the law the quantitative changes in a system eventually pass into qualitative changes. Not surprisingly, this law could also explain the advances that have been taking places during the last decade in the computer industry.

The quantitative advances in the computer industry were predicted and formalized by the Moore's law. The law predicted the doubling of transistor densities every 18 to 24 months. Coupled with the increase of processor frequencies these led manufacturers to produce faster and more powerful processors every year. But these advances inevitably came to their end at some point in time. Growing disparity of speed between the CPU and the memory could not further improve the speed of computation. Additionally, increasing density of transistors leads to prohibitively high levels of power consumption, heat and power leakage. This is a prove that quantitative advances in hardware technologies could not be further a driving force in the computer industry.

To overcome these problems chip manufacturers have moved from single processor systems to multiprocessor or parallel systems. This allows to run processors on lower frequencies and combine general purpose processors with dedicated processors leading to heterogeneous systems with more optimized usage of transistors. The world of computers moved from sequential computing to parallel computing, or, in other words, the quantitative advances have been transformed into qualitative advances.

Although, we are witnessing the emergence of parallel (multi-core and multi-processor) systems in all markets: from general-purpose computing to embedded systems, e.g., multimedia systems, game consoles and all sorts of mobile devices, the transition from sequential to parallel computing is far from trivial. To satisfy emerging applications requirements, the multiprocessor embedded systems must be pro-

grammed in a way that the available parallelism is revealed and exploited efficiently. However, the programming of a multiprocessor system is a challenging, error-prone, and time consuming task as it involves the partitioning of programs, and consequently, synchronization of different program partitions.

In recent years, a lot of attention has been paid to the design of parallel systems. However, insufficient attention has been paid to the development of concepts, methodologies, and tools for efficient programming of such systems. Therefore, the programming still remains a major difficulty and challenge [1]. Today, system designers experience significant difficulties in programming parallel systems because the way an application is specified by an application developer, typically as a *sequential* program using a *sequential* model of computation (MoC), does not match the way multiprocessor systems operate, i.e., multiple cores run (possibly) in parallel.

If an application is specified using a *parallel* MoC, then the mapping of this application onto a multiprocessor system can be done in a systematic and transparent way by using a disciplined approach [2]. Using a parallel MoC facilitates the programming of parallel multiprocessor systems because a parallel MoC makes the parallelism available in an application and the communication between the application tasks explicit. Unfortunately, specifying an application using a parallel MoC is very difficult as the application developers i) have to be familiar with a particular parallel MoC; ii) have to study the application in order to identify possible parallelism that is available and to reveal it by using the parallel model.

To relieve the designer from all these difficulties, the *pn* compiler [3] was introduced. It implements techniques for automated parallelization of static affine nested loop programs (SANLP) written in a subset of *C*-language equivalent to the program model presented in [4] into input-output equivalent Polyhedral Process Network (PPN) descriptions. In the *pn* partitioning strategy, a process is created for every statement and function call found in the top-level of the program. In this way, the designers have control over the granularity of the created partitions.

```
1 parameter N 10 100;

2  for j = 1 to 6*N-3,
3    A[j] = Func1()
4  endfor

5  for j = 0 to N,
6    for i = j to 3*j-2,
7      if( i+j < 4*N-6 )
8        A[i] = Func2( A[2*i-1], A[2*i+1] )
9      endif
10     Func3( A[i] )
11   endfor
12 endfor
```

Figure 1.1: Pseudo code of a SANLP.

An example of a SANLP is given in Figure 1.1. In this and following figures in this dissertation, the examples will be given in pseudo code which semantics is equivalent to the semantics of the program model presented in [4]. We decided to use the pseudo code syntax instead of *C*-language syntax in order to avoid syntax clutter

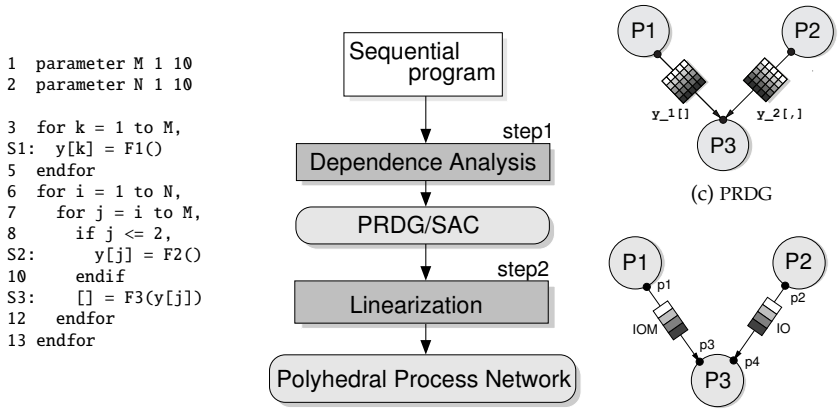
of *C*-language and emphasize that input programs considered in this dissertation are not language specific. A SANLP in Figure 1.1 consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops do not have to be perfectly nested. All lower and upper bounds of the loops as well as all expressions in conditions and array accesses have to be affine functions of enclosing loop iterators and static parameters. The parameters are symbolic constants, i.e., their values can not change during the execution of the program. Rather, parameter values determine different program instances. In addition, data communication between function calls must be explicit. For example, see function $Func2()$ at line 8 which accepts 2 elements of array A as input arguments. Providing just a pointer to array A in this case is not allowed. The above restrictions allow a compact mathematical representation of a SANLP using the well-known polyhedral model [5]. The SANLPs can be converted in an automated way into Polyhedral Process Networks [3].

The target Polyhedral Process Networks (PPNs) [6] is a special case of the Kahn Process Networks (KPNs) [7] model of computation. A PPN consists of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels using a blocking read/write on an empty/full FIFO as synchronization mechanism. In addition, everything about the execution of a PPN is known at compile-time. The latter enables techniques for modeling, analysis, and SW/HW synthesis in a systematic and automated way, and allows the calculation of buffer sizes that guarantee deadlock-free execution. In comparison, computing buffer sizes is not possible for the more general KPN model. We are interested in the process network model because it provides a sound formalism, well suited for capturing and modeling of data-flow dominated applications in the realm of multimedia, imaging, and signal processing, that naturally contain tasks communicating via streams of data. Moreover, it has been already shown that process networks allow effective and efficient mappings of streaming applications to certain parallel execution platforms [8–13].

1.1 Automatic Derivation of Polyhedral Process Networks

In this section we briefly recall the systematic parallelization approach [14] used to derive Polyhedral Process Network from static sequential programs. This overview will help to understand the contributions of this dissertation.

The approach is illustrated in Figure 1.2(b). It starts with an application written as a sequential program similar to the one depicted in Figure 1.1. In many cases completely sequential execution can be relaxed without compromising the correctness of the execution. That is, the order of the program statements can be rearranged without changing the program functionality under the ordering constraints. In fact, the possibility to rearrange statements is actually the possibility to execute them in parallel. Ordering constraints are dictated by the data dependency relations existing in the sequential program. Therefore, the first main step of the parallelization



(a) SANLP Program (b) Parallelization approach (c) PRDG (d) PPN

Figure 1.2: An example of a Static Affine Nested Loop Program, approach that translates SANLPs into PPNs, intermediate representation Polyhedral Reduced Dependence Graph and resulting PPN.

approach depicted in Figure 1.2(b) is to perform *data dependence* analysis between evaluations of statements. The analysis helps to extract the dependent memory accesses and present an initial program in a form where data dependencies are made explicit. Thus, the initial program is translated into the Single Assignment Code (SAC) [4] form or its analogous form called Polyhedral Reduced Dependence Graph (PRDG) [15] which is a compact mathematical representation of the dependency relations in terms of polyhedra. In PRDG the nodes represent statements of the initial program and the edges represent data dependencies. The PRDG model still exploits (multi-)dimensional arrays for data communication. However, the target model, Polyhedral Process Networks, requires FIFO channels as communication medium. Therefore, another step is needed that converts multi-dimensional memory accesses into managed dataflow over FIFO queues. This step called *Linearization* constitutes the second main step of the approach shown in Figure 1.2(b). In the following, we give more detailed overview of these steps.

1.1.1 Dependence Analysis

The goal of the *Dependence Analysis* step 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 particular 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 most relevant algorithm to perform the analysis is the Exact Array Dataflow Anal-

ysis (EADA) [4]. This algorithm considers all pairs of statements where one writes and the other reads from possibly the same memory addresses. For each pair a special system of linear (in)equalities is built. This system defines in an affine form the conditions when two statements are data dependent. The formal definition of this system of linear (in)equalities will be given in Chapter 2. For example, Table 1.1 depicts two systems of linear inequalities $\mathbf{Q}_{\text{S1S3}}((i_3, j_3))$ and $\mathbf{Q}_{\text{S2S3}}((i_3, j_3))$ for pairs S1S3 and S2S3, respectively. Note that with (i_3, j_3), iteration vector (i, j) of statement $S3$ is denoted. Similarly, with (i_2, j_2) and k iteration vectors of statements S1 and S3 are denoted, respectively.

The meanings of constraints (c1)–(c3) are the following. The first constraint (c1) states that the source iteration has to exist, i.e., it has to belong to the iteration domain of a potential source statement. The constraint (c2) specifies that if there is a dependency between evaluation of two statements, both have to access the same array element. The (c3) constraint determines an order of operations, i.e., writing to the array cell in one statement should be performed *before* reading from the same array cell in the other statement. The order of operations is defined by the sequential order of the initial program.

$\mathbf{Q}_{\text{S1S3}}((i_3, j_3))$	$\mathbf{Q}_{\text{S2S3}}((i_3, j_3))$	
$1 \leq k \leq M$	$1 \leq i_2 \leq N \wedge i_2 \leq j_2 \leq M \wedge$ $j_2 \leq 2$	(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 1.1: Systems used in dependence analysis between statements S1S3 and S2S3 in the program shown in Figure 1.2(a).

Having these systems formulated, the data dependence algorithm finds the lexicographical maximum between all vectors satisfying the systems. Finding a lexicographical maximum means to solve the Parametric Integer Linear Problems (PILPs), where **objective** is the *lexmax* function **subjected to** conditions stated by systems $\mathbf{Q}()$. The solution to find the maximum point for a given convex domain defined by the systems 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]. Finally, the source operation for the data read by statement S3 is:

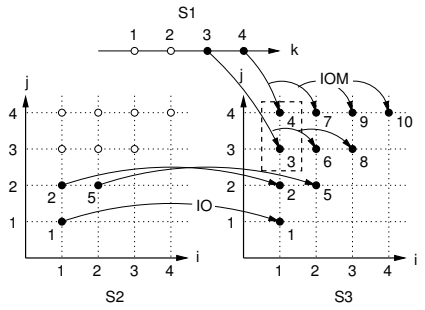
$$\begin{array}{lll} \textbf{if} \quad j_3 \leq 2 & \textbf{then} & \langle S2, (i_3, j_3)\rangle \\ & \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \tag{1.1}$$

The solution above shows that the source of the data for statement $S3$ of the program in Figure 1.2(a) can be from two different statements. The source is statement

$S1$ when the iterator j of $S3$ is greater than 2. Otherwise, the source is statement $S2$. The result of the analysis forms the dependency relations between iterations of statements writing/reading to/from the array.

```
1   for k = 1 to 4,
S1:   y_1[k] = F1()
3   end
4   for i = 1 to 4,
5     for j = i to 4,
6       if j <= 2,
S2:       y_2[i,j] = F2()
8       end
9       if j <= 2,
10        in_0 = y_2[i,j]
11      else
12        in_0 = y_1[j]
13      end
S3:     [] = F3(in_0)
15    end
16  end
```

(a) SAC code

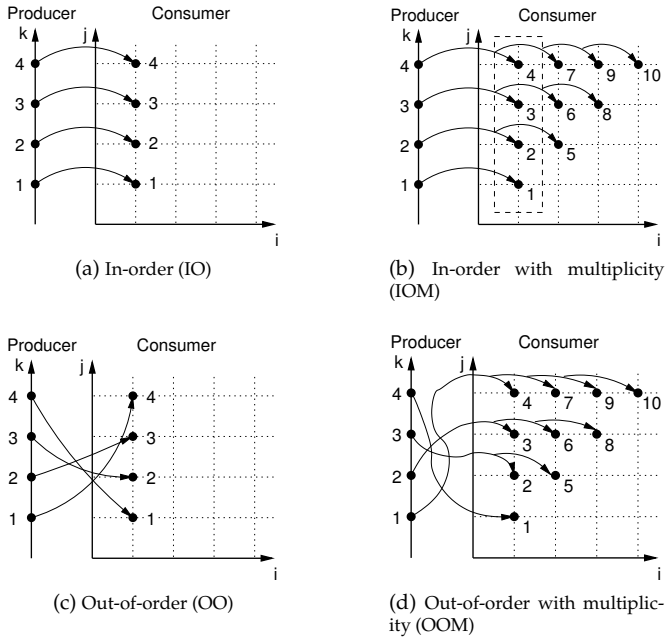


(b) Data dependencies

Figure 1.3: Single Assignment Code form and data dependencies relations in the program depicted in Figure 1.2(a).

For illustrative purposes, the same result can be depicted in a graphical form as shown in Figure 1.3(b). In the figure the dependencies are found for values of program's parameters M and N equal to 4. The coordinate systems show the iteration domain of statements $S1$, $S2$ and $S3$. The points on the coordinate systems designate the evaluations of statements $S1$, $S2$ and $S3$, and the arrows reflect the data dependency relations.

Based on the results of the dependence analysis, the initial sequential program can be translated into the Single Assignment Code (SAC) form which is introduced in [4]. The SAC program is functionally equivalent to the initial program, with the difference that every variable is written exactly once. The latter ensures that all data dependencies are explicitly revealed and respected. The SAC form of the initial program shown in Figure 1.2(a) is illustrated in Figure 1.3(a). In this SAC program, the original array `y[]` is substituted with multi-dimensional dedicated arrays `y_1[]` and `y_2[,]`, and control at lines 6 and 9 is added in order to respect the original data dependency relations. The SAC form of the initial program is easily convertible to the Polyhedral Reduced Dependence Graph (PRDG) [15]. The PRDG is a graph where nodes represent computation and edges represent communication. Nodes communicate point-to-point via unique dedicated multi-dimensional arrays that respect the original data dependencies. An example of a PRDG derived from the program shown in Figure 1.2(a) is illustrated in Figure 1.2(c). It consists of three processes that correspond to the statements of the SAC program shown in Figure 1.3(a), and two arrays that correspond to the arrays `y_1[]` and `y_2[,]` of the same program. It is important to note that rather than introducing a node for each iteration of a statement in a SAC, in PRDG, a node specifies a *regular set* of all iterations of a statement in a SAC program. This regular set is defined in terms of *polyhedra*. The definition of polyhedra and its relation to affine nested loop programs will be given



(a) In-order (IO)

(b) In-order with multiplicity (IOM)

(c) Out-of-order (OO)

(d) Out-of-order with multiplicity (OOM)

Figure 1.4: Types of communication models in a PPN.

in Chapter 2.

To summarize, the *Data Dependence* step of the parallelization approach shown in Figure 1.2(b) translates the initial sequential program into its functionally equivalent PRDG which specifies a program in terms of topology, behavior and geometry. The geometrical polyhedral specification makes this model useful for different transformations [20]. The explicit separation between communication, computation and geometrical characterization allows for translation from communication via shared memory to FIFO channels which constitutes the *Linearization* step of the parallelization approach.

1.1.2 Linearization

In a PRDG, the storage structure of the initial program is transformed such that each Producer/Consumer (P/C) pair of nodes communicates over dedicated multi-dimensional memory arrays as shown in Figure 1.2(c). However, in the target PPN model, communication is required to be done via FIFO channels instead of multi-dimensional arrays because, for example, such communication allows for simple implementation of data streams in software and hardware. The *Linearization* step of the parallelization approach replaces all such multi-dimensional arrays with FIFO channels.

However, due to the particular way the data flows from a Producer node to a Consumer node, mapping array communication onto FIFO channels requires complex address generators, especially if the arrays have multiple dimensions. Therefore, the Linearization also solves the Communication Model Identification (CMI) problem, which investigates communication characteristics of each P/C pair of nodes in a PRDG. The CMI problem is an optimization problem that allows to identify communication behavior of a FIFO channel that is cheaper for realization. We explain this in the following example.

Every FIFO channel implemented as a point-to-point communication has one Producer and one Consumer node or processes, thereby forming a Producer/Consumer pair (P/C pair). An example of two P/C pairs P1/P3 and P2/P3 is shown in Figure 1.2(d).

In any point-to-point communication, the firings of the Producer process generate data tokens in a certain order. We call it the production order. The tokens are sent to the Consumer process over the FIFO channel. In order to fire, the Consumer process needs the data tokens in a certain order. We call it the consumption order. When the production and consumption orders are the same, we say that the P/C pair communication is *in-order*. Otherwise, the P/C pair communication is *out-of order*. Consider, for example, Figure 1.4(a). It depicts the Producer and Consumer processes, where the points on the coordinate systems designate the firings of the processes and the arrows reflect the data dependencies between firings. The numbers at the points show the production/consumption orders. Figure 1.4(a) shows that the production order of the Producer process coincides with the consumption order of the Consumer process. This is an example of *in-order* communication in a P/C pair. Similarly, Figure 1.4(c) illustrates that the consumption order of the Consumer is reversed to the production order of the Producer process. This is an example of *out-of-order* communication.

In a P/C pair, it may occur that the Consumer process may need to reuse in future firings a token that has just been received from the Producer process. In such case we say that the P/C pair communication has a *multiplicity*. For example, consider the firing of the Producer and Consumer depicted in Figure 1.4(b). The production and consumption orders are the same, thus, the P/C pair communication is *in-order*. Additionally, we may notice, for example, that the data token needed for firing 3 of the Consumer process, will be needed on firings 6 and 8. Thus, the P/C pair communication has a multiplicity. Likewise, a P/C pair communication may be *out-of-order* and has a *multiplicity*. An example of such P/C pair communication is depicted in Figure 1.4(d). The four different types of P/C pair communication described above, determine four communication models between processes. They are: *in-order (IO)*, *out-of-order (OO)*, *in-order with multiplicity (IOM)* and *out-of-order with multiplicity (OOM)*. The communication of a P/C pair belongs to one of these four types only.

According to the explanations given above, the communication models between nodes formed by statements S1/S3 and S2/S3 in Figure 1.3(b) are IOM and IO, respectively.

```
1  parameter M 1 10;
2  parameter N 1 10

3  for k = 1 to M,
S1:  y[k] = F1()
5  endfor
6  for i = 1 to N,
7    for j = i to M,
C:     if y[j] <= 2,
S2:      y[j] = F2()
10     endif
S3:    [] = F3(y[j])
12   endfor
13 endfor
```

(a) Weakly Dynamic Program (WDP).

```
1  parameter M 1 10;
2  parameter N 1 10

3  for k = 1 to M,
S1:  y[k] = F1()
5  endfor
6   for j = 1 to N,
7     for i = 1 to f(...),
S1:     y[i] = F1()
9     endfor
10   endfor

S2: [...] = F2( y[5] )
```

(b) Affine program with dynamic loop bounds (`Dynloop`).

```
1 parameter M 1 10;
2 parameter N 1 10

3  for k = 1 to M,
S1:  y[k] = F1()
5  endfor
6  for i = 1 to N,
S1:  y[i] = F1()
8    while (...),
S2:    y[i] = F2()
10    endwhile
S3:  [] = F3( y[i] )
12  endfor
```

(c) Affine program with while-loop (WLAP)

Figure 1.5: Examples of WDP, `Dynloop` and WLAP programs. The differences are that in WDP program, there is a dynamic `if`-condition at line C; in `Dynloop`, the upper bound of for-loop *i* at line 7 is data-dependent; in WLAP program, the second loop at line 8 is a while-loop.

In order to implement a PPN, all communication models have to be realized over a FIFO channel. The *in-order* models can be implemented with a FIFO in a straightforward way, as the order of writing into the FIFO channel and the order of reading from it are the same. The *out-of-order* models would require a FIFO channel augmented with a controller implementing the reordering. In a similar manner, the models with *multiplicity* would require a FIFO channel with additional memory to store the tokens which will be reused later.

The difference in realization puts the communication models into a hierarchy. The realization of the *OOM* model is the most general, as it is built of all components present in the realizations of the other models. In other words, all P/C pair communication models can be implemented with the realization of the *OOM* model. However, the more general a realization is, the more resources it needs and more run-time overhead is introduced. More importantly, the *IO*,*OO* and *IOM* communication models might be implemented with simpler realization.

Thus, we see that in order to parallelize a sequential application two important steps, *Dependence Analysis* and *Linearization*, should be addressed.

1.2 Problem statement

Many scientific, matrix computation, and signal processing applications can be specified as static affine nested loop programs (SANLPs), and therefore, the *pn* compiler [3], briefly described in Section 1.1, can be used to derive equivalent parallel PPN specifications. However, many multimedia applications such as MPEG coders/decoders, Smart Cameras, adaptive filters, iterative algorithms, etc. have adaptive

and dynamic behavior which can not be expressed as static programs as SANLPs.

In order to handle such dynamic applications, in this dissertation we address an important question: whether some of the static restrictions of the SANLPs can be relaxed while keeping the ability to perform compile-time analysis and to derive PPNs in an automated way. Achieving this will significantly extend the range of applications that can be parallelized in an automated way.

By studying different dynamic applications we distinguished three relaxations to SANLP programs that would allow one to specify dynamic applications as sequential programs.

The first relaxation:

I. dynamic `if`-conditions are allowed in a program.

An example of an application with this relaxation is depicted in Figure 1.5(a). Note, that the `if`-statement at line C has a dynamic condition "`y[j] <= 2`". This condition is dynamic because it depends on the variable `y[j]` whose value is determined during program execution. As a real-life example of an application that contains such relaxation, the Motion JPEG (M-JPEG) encoder [21–23] can be considered.

The second relaxation:

II. for-loops with dynamic bounds are allowed in a program.

An example of an application with this relaxation is depicted in Figure 1.5(b). Note, that the upper bound of for-loop `i` at line 7 is an arbitrary function `f(...)`. As an example of a real-life application containing for-loops with dynamic bounds an application from the smart cameras domain called Low Speed Obstacle Detection (LSOD) [24] can be considered. The detailed analysis of this application will be given in Chapter 6.

The third relaxation:

III. while-loops are allowed in a program.

An example of an application with this relaxation is depicted in Figure 1.5(c). Note, that the loop at line 8 is a while-loop. An example of a real-life application containing while loop is the application from the signal processing domain called Adaptive Beamforming (AB) [25].

In the rest of this dissertation, a program that contains either of these three relaxation will be called a *dynamic program*.

In [21], the first relaxation has been considered, i.e., how to translate affine nested-loop programs with dynamic `if`-conditions into input-output equivalent PPNs in

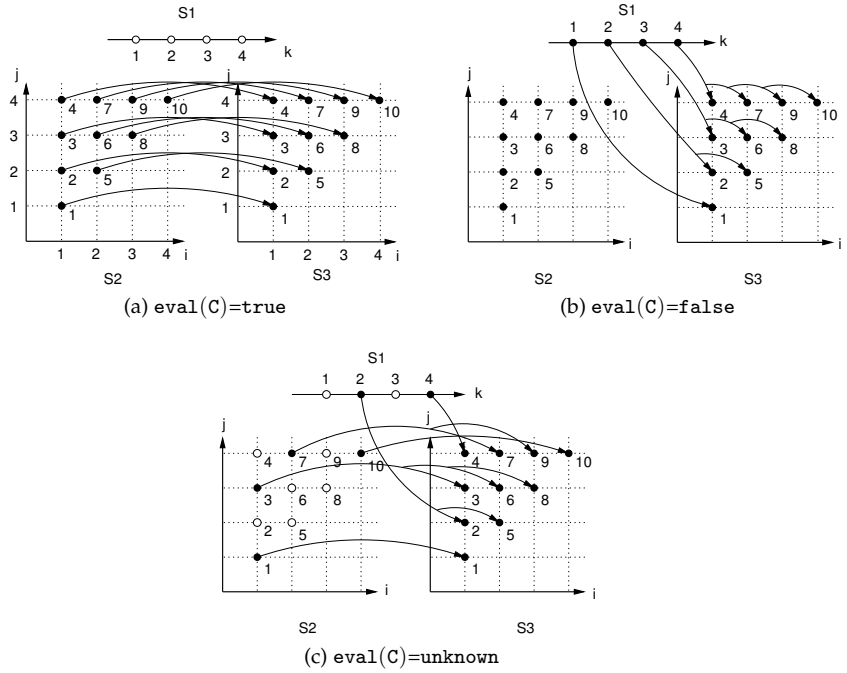


Figure 1.6: Dependency relations in a dynamic program.

an automated way. In this dissertation, we consider the other two more difficult relaxations. Finally, we formulate the problem which is solved in this dissertation:

to develop an automated procedure for translation of affine programs with relaxations II and III into input-output equivalent Polyhedral Process Networks.

1.3 Motivation and challenges

The overall challenge of deriving a PPN from a dynamic program is how to deal with uncertainties introduced by the relaxations presented in Section 1.2. Below, we demonstrate that: 1) an exact data dependence analysis (EADA) and *exact* communication model identification (CMI) in PPNs derived from dynamic programs are not possible at compile-time; 2) the existing approach used for PPN derivation presented in Section 1.1 cannot be used to identify communication models and translate dynamic programs into PPNs; 3) nevertheless, at compile-time, for any P/C pair it is possible to identify the most general communication model which can be used to realize all possible data dependency patterns that may occur in different instances of the dynamic program.

Consider, for example, the dynamic program shown in Figure 1.5(a) which contains dynamic relaxation I. The program has a dynamic condition at line C evaluating some run-time data. Depending on the evaluation of the `if`-condition, either statement S1 or S2 produces the data for every firing of statement S3.

Figure 1.6 depicts data dependency relations between statements S1, S2 and S3 in three instances of this program for $M = N = 4$, where an instance of a dynamic program is an evaluation of the program with a particular input dataset. Figure 1.6 illustrates iteration domains of statements S1, S2 and S3, where the points on the coordinate systems designate the evaluations of statements and the arrows reflect the data dependencies between evaluations. The numbers at the points show the lexicographical order of statement evaluations.

Assume, first, that the condition at line C always evaluates to *true*, and, thus, all the data needed by statement S3 is produced by statement S2 only (see Figure 1.6(a)). The opposite case is when the condition at line C always evaluates to *false*. Depicted in Figure 1.6(b), this time, relations exist between statement S1 and S3 only. In general, however, the result of condition evaluation at line C is arbitrary and unknown at compile-time. An example of this case is shown in Figure 1.6(c). In this case, on some firings, the data needed by statement S3 is produced by statement S1, on other firings by statement S2.

The three examples of data dependency relations illustrated in Figure 1.6 show the difference of dependency patterns between dynamic and static programs. In static programs, different instances of a program correspond to one and the same single dependency pattern which is known at compile-time. In dynamic programs, data dependency patterns correspond to different *instances* of a dynamic program, and are unknown at compile time. This also means that data dependency patterns in a dynamic program cannot be determined at compile-time by the exact array dependence analysis.

We can illustrate the same idea by using the description of the exact dependence analysis presented in Section 1.1.1. Recall, that this analysis constitutes Step 1 of the parallelization approach depicted in Figure 1.2(b). In order to determine data dependency patterns at compile-time the data dependence analysis has to be performed on a initial program. The dependence analysis algorithm builds a system of linear inequalities similar to one shown in Table 1.1. Consider, for example, the dynamic program in Figure 1.5(a), and let us build the system for pair S2S3:

Then, the data dependence algorithm finds the lexicographical maximum between all vectors satisfying the systems by solving Parametric Integer Linear Problems problem. However, in the system shown in Table 1.2 constraint $(c1)$ which specifies the domain of the source iteration is not a convex domain as it contains dynamic `if`-condition $y[j] <= 2$. Therefore, the exact data dependence analysis presented in Section 1.1.1 cannot be applicable to dynamic programs.

The same reasoning applies to the dynamic programs with the other two relaxations II and III. Overall, this shows that the approach presented in Section 1.1 cannot handle the dynamic programs shown in Figure 1.5.

$\mathbf{Q}_{S2S3}((i_3, j_3))$	
$1 \leq i_2 \leq N \wedge i_2 \leq j_2 \leq M \wedge$ $y[j_2] \leq 2$	(c1)
$j_2 = j_3$	(c2)
$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

Table 1.2: The system used to perform exact dependence analysis between statements S2S3 in the program shown in Figure 1.5(a).

The inability to determine the exact data dependency relations in a program makes the exact communication model identification impossible. Nevertheless, still we can analytically identify at compile-time the communication models of a P/C pair in all possible instances of a dynamic program. Based on this information, we can realize the communication of a P/C pair with the most general communication model which implements all possible data dependency relations. For example, we may observe in Figure 1.6 that the production/consumption orders in S1/S3 and S2/S3 pairs are the same. Thus, the communication in all P/C pairs is *in-order*. Moreover, in some program instances a multiplicity in the communication is possible and, according to the realization hierarchy of communication models, the most general model for S1/S3 and S2/S3 pairs is *IOM*. Therefore, we can implement the communication in S1/S3 and S2/S3 pairs as *IOM* model.

In order to solve the problem addressed in this dissertation and formulated in Section 1.2, we split the problem in 2 issues. The first issue is formulated as follows:

- **Issue I**: dynamic programs with the relaxations presented in Section 1.2 require different parallelization approach in translating them into PPNs compared to the approach presented in Section 1.1. Loop iteration domains and the overall dataflow is unknown at compile-time in dynamic programs. [21] considered relaxation I, i.e., how to translate affine nested-loop programs with dynamic `if`-conditions into input-output equivalent PPNs in an automated way. The main research topic of this dissertation is how to translate dynamic programs with relaxation II and III into PPNs.

The first issue calls for a novel parallelization approach for dynamic programs. The second issue below addresses a novel approach for implementing a PPN.

- **Issue II**: we demonstrated that an exact communication model identification in PPNs derived from dynamic programs is not possible at compile-time. Therefore, we address the problem of communication model identification in PPNs derived from dynamic programs. This issue is an important optimization problem as it allows to identify communication models with simpler realization.

1.4 Research Contributions

The work presented in this dissertation focuses on the derivation of Polyhedral Process Networks specifications from dynamic programs. Below, we list the contributions delivered by this dissertation.

- **Contribution I [26, 27]:** a first approach for automated translation of affine nested-loop programs with dynamic loop bounds (Dynloop) into input-output equivalent Polyhedral Process Networks. In addition, we present a method for analyzing the execution overhead introduced in the PPNs derived from programs with dynamic loop bounds.
- **Contribution II [28]:** a first approach for automated translation of affine nested-loop programs with *while*-loops (WLAP) into input-output equivalent Polyhedral Process Networks.
- **Contribution III [29]:** we present a formal procedure for communication models identification in Polyhedral Process Networks derived from the dynamic programs introduced in Section 1.2.

To address the first issue defined in Section 1.3, Contributions I and II of this dissertation devise a compile-time automated procedure that can be used to derive a PPN from dynamic programs with relaxations II and III presented in Section 1.2. By our third contribution we address the second issue: we introduce a novel procedure for Communication Model Identification in PPNs derived from dynamic programs.

1.5 Related Work

The work presented in this dissertation is directly related to previous works on systematic and automated derivation of process networks from affine nested loops programs initiated by Rijpkema et al. [15, 30]. Further, Turjan et al. [31] proposed an automated derivation of process networks from *static* affine nested loop programs (SANLPs). In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Stefanov [21] further developed a procedure of process network derivation from more relaxed class of affine nested loop programs called *Weakly Dynamic Programs* (WDPs). In this class of affine nested loops programs, the conditions in control structures might be dependent on some information that is unknown at compile-time and may change at run-time. In contrast, this dissertation deals with more general class of applications, i.e., affine nested loop programs with loop bounds (`Dynloop`) that unknown at compile-rime and determined at run-time, and applications containing while-loops (WLAP).

In the context of automatic parallelization of sequential programs research has been done on approaches to convert a nested loop program to an equivalent program

which is in a single-assignment form, i.e., a program in which every memory cell is written at most once. Such program is easier to be analyzed and parallelized efficiently. The work of Knobe and Sarkar [32], Feautrier et al. [33] and the work of Griebl, Lengauer and Collard [34–36] on this topic are directly related to the first step of our approaches presented in Chapters 3 and 4 of this dissertation. This is because in this step we propose an approach to convert dynamic programs into a single-assignment form which we call dynamic Single Assignment Code (dSAC). The relations are explained below.

Knobe and Sarkar [32] proposed an approach to convert a nested loop program to a single-assignment form that they call Array Static Single Assignment (ASSA). Their approach is more general than our approach in the sense that the class of nested loop programs which they can convert to their ASSA includes classes of dynamic programs considered in this dissertation which we can convert to our dSAC. However, when Dynloop and WLAP programs are considered, our approach is more efficient compared to their approach in the sense that dSAC is a more efficient single-assignment form in terms of code lines and memory usage compared to their ASSA form. This is because in our approach a dependence analysis is performed at compile-time before the corresponding dSAC is generated. This compile-time dependence analysis, called fuzzy array data-flow analysis (FADA) [37, 38], allows an efficient code generation. The approach of Knobe and Sarkar does not perform any dependence analysis at compile-time. Instead, the dependence analysis is performed at run-time by placing a special code called ϕ functions and @ arrays in their ASSA, thereby making their approach more general. The ϕ functions and @ arrays introduce significant code overhead because in many cases unnecessary ϕ functions and @ arrays are placed in the ASSA, thereby making the ASSA form very inefficient in terms of code lines and memory usage compared to our dSAC.

The work of Feautrier et al. in the context of the PAF parallelizer [39] describes an approach to convert nested loop programs similar to our dynamic programs into a single-assignment form called SA. Their approach is based on performing a fuzzy array data-flow analysis (FADA) at compile-time before generating the SA. The result of this FADA analysis is implemented by ϕ functions placed in their SA during code generation. The ϕ functions depend on parameters whose values have to be set dynamically at run-time in order to preserve the original data-flow when the control flow cannot be predicted at compile-time. The work of Feautrier et al. lacks a general approach to set the values of the parameters at run-time. The work described above relates to our approach for converting dynamic programs to a dSAC in the sense that we also perform a FADA analysis at compile-time and we also place a code with parameters in our dSAC similar to the ϕ functions but our code is more efficient. Also, the difference is that we have developed a very simple general approach to set the values of the parameters at run-time. This approach is presented in Chapter 3.

Griebl, Lengauer and Collard [34–36] addressed the problem of parallelization of while-loops similar to our work. Similar to our approach, they perform array dataflow analysis to expose data dependencies in an explicit way. Subsequently, they use space-time restructuring techniques to generate the code for speculative execution

or software pipelining. Generally unscannable execution space that a while-loop provides, they scan with the help of run-time computable predicates, that are also used for detection of while-loops' termination. Besides introducing an overhead at run-time, these predicates limit the applicability of their approach to shared memory systems. In contrast, our parallelization approach targets multiprocessor systems with distributed memory.

Apart from the idea to convert a nested loop program to an equivalent program in a single-assignment form, in the context of automatic parallelization of dynamic sequential programs, there are a number of other efforts that were made.

Raman et al. [40] devise the Parallel-Stage Decoupled Software Pipelining (PS-DSWP) multi-threading technique to extract pipeline parallelism from codes with irregular, pointer-based memory accesses and arbitrary control flow, which generally include while-loops. A parallel-stage allows to obtain pipeline parallelism from some stages executed in a DOALL fashion. In contrast, our approach supports also task- and data-level parallelism besides the pipeline- and iteration-level parallelism. Moreover, we can generate parallel code for multi-processor systems with distributed memory.

The LooPo compiler [41] deals with parallelization of more general class of nested loop program than the class we consider in this dissertation. It includes nested loop programs with unscannable execution spaces which boundaries are determined at run-time. The proposed parallelization procedure is based on run-time detection of executed statements as well as detection of program termination [42]. In contrast to [41], we use FADA and perform approximated dependence analysis at compile-time. Moreover, we do as much as possible analysis at compile-time, thereby reducing the run-time overhead significantly.

A different approach is taken by Benabderrahmane et al. [43] where they embed the control and exit predicates to the general data-dependent control-flow programs. This predicates are used instead of data dependent control structures and while-loops as first-class citizens of the algebraic representation. Subsequently, a polyhedral representation is derived and code generation is performed from static program analysis. In this approach, hiding all dynamism (dynamic loop bounds, while-loops) in algebraic representations also diminishes the parallelism available in the initial program as less information is visible for analysis. By contrast, our technique exposes and utilizes all available parallelism.

Rauchwerger et al. [44] focused on parallelizing while-loops that are defined by one or more recurrences that can be detected at compile-time; a reminder that can be either analyzed statically or is unknown at compile-time; and one or more termination conditions. Although, they were able to parallelize a while-loop involving linked lists traversal, it is not shown how they would tackle more general while-loops, which we consider in this dissertation.

Bijlsma [45, 46] and Geuns [47] approach the problem of while-loops parallelization by considering an initial program with while-loops being in the local single assignment (LSA) form where all data dependencies are explicit. They implement the ex-

plicit data dependencies using circular buffers with overlapped read and write windows. Specifying a program in a LSA form can be very time consuming and error prone process because the system designer has to do the dependence analysis manually. We find this a very a limitation of their work. By contrast, our approach uses an automatic data-dependence analysis procedure which relieves the designer from the very difficult task to do the manual dependence analysis.

In the context of communication model identification in Process Networks, to the best of our knowledge, not much attention has been devoted to the problem of automatic communication model identification. An automatic procedure exists for communication model identification while translating *static* affine nested loop programs (SANLP) into functionally equivalent PPNs [31]. In this dissertation, we develop an extension to this procedure which identifies communication models in Polyhedral Process Networks derived from the dynamic programs introduced in Section 1.2.

1.6 Dissertation Outline

The remaining part of this dissertation is organized a follows. In Chapter 2, we first introduce notations and terminology that will be used throughout the dissertation. Further, we present theory describing the Polyhedral Model and show how this model can be extracted from applications considered in this dissertation.

In Chapter 3, we present a first approach for automated translation of affine nested loop programs with dynamic loop bounds (`Dynloop`) into input-output equivalent Polyhedral Process Networks. The chapter describes in great details the models, methods, and techniques we have developed and used in the approach. First, we describe the techniques and procedures involved in the conversion of a Dynloop to our dynamic Single Assignment Code (dSAC). Second, we demonstrate how the free parameters introduced by FADA analysis are assigned in dSAC using control arrays. Third, we show how the topology of the corresponding PPN is derived, as well as the code executed in each process. Moreover, we demonstrate how the buffer sizes of FIFO channels are computed that guarantee a deadlock-free execution of a PPN.

In Chapter 4, we present a first approach for translation of *affine nested loop* programs with *while*-loops (WLAP) into input-output equivalent PPNs. In this chapter we describes in great details the models, methods, and techniques we have developed and used in the approach.

In Chapter 5, we present a formal procedure for identifying communication models in process networks derived from the dynamic programs introduced in Section 1.2. We formulate two problems from integer linear problems domain that allow us to identify the communication models presented in Section 1.1.2.

In Chapter 6, we present a case study that we conducted in order to validate and evaluate our approach presented in Chapter 3. This case study presents a real-life industrially relevant application. We will present a comprehensive analysis and re-

port the results we obtained in this case study.

Finally, we conclude this dissertation in Chapter 7 with a summary of the presented research work along with some concluding remarks.

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.

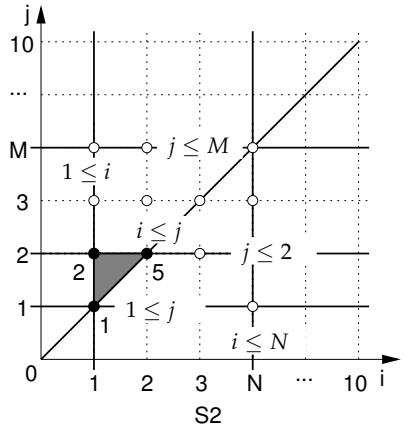


Figure 2.1: Geometrical representation of iteration domain of statement S2 in the program depicted in Figure 1.2(a).

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).

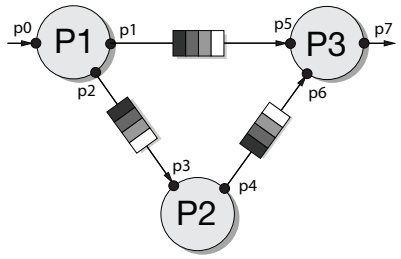
2.3 Polyhedral Process Networks

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.

Chapter 3

Automated Generation of Polyhedral Process Networks from Affine Nested-Loop Programs with Dynamic Loop Bounds

In this chapter, we introduce a first approach for automated translation of affine nested loop programs which contain relaxation I, i.e., dynamic loop bounds (`Dynloop`), into input-output equivalent Polyhedral Process Networks (PPNs). We developed this approach in order to address an important question: whether static restrictions on loop bounds in Static Affine Nested Loop Programs (SANLPs) can be relaxed while keeping the ability to perform compile-time analysis and to derive PPNs in an automated way. Achieving this would significantly extend the range of applications that can be parallelized in an automated way.

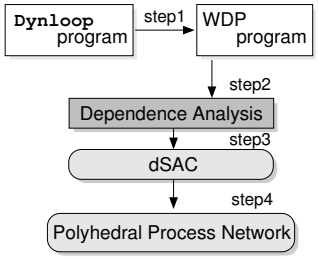
Recall, that in Section 1.1 we briefly introduced the main steps needed to translate a *static* sequential application into a PPN. Additionally, in Section 1.3 we showed that this approach cannot be used on dynamic applications. In this chapter we develop a new approach elaborating in more detail on the new models and techniques that are used in parallelization of a `Dynloop` program.

```
1   parameter N 1 10;

2   for j = 1 to N,
3     for i = 1 to f(...),
S1:     y[ i ] = F1()
5     endfor
6   endfor

S2: [...] = F2( y[5] )
```

(a) Dynloop program



(b) Parallelization approach

Figure 3.1: An example of a Dynloop program and an approach that translates Dynloop into PPNs.

3.1 Solution Overview

The high-level overview of the approach is illustrated in Figure 3.1(b). It starts with an application written as a sequential program that has dynamic loop bounds similar to one depicted in Figure 3.1(a). We have found out that a Dynloop program can be formally represented as a Weakly Dynamic Program (WDP). Therefore, in the first step of the approach, the initial Dynloop program is represented as a WDP. For WDP programs we can employ the Fuzzy Array Dependence Analysis (FADA) [37, 38] technique, described in Section 2.5. The analysis, which constitutes the second step of the approach, helps to extract the dependent memory accesses and present an initial program in a form where data dependencies are made explicit. In Section 1.3 we showed that in a Dynloop program *exact* data dependency patterns are unknown at compile time. The FADA analysis allows to parameterize (or approximate) such data dependency patterns with parameters which values are determined at run-time. In the third step, based on the results of the FADA dependence analysis, the initial sequential program is translated into a *dynamic* Single Assignment Code (dSAC) representation of the WDP program. The dSAC was proposed in [21] as an extension of the Single Assignment Code [4]. A dSAC program is input-output equivalent to the corresponding WDP and it has the property that every data variable or an array element is written *at most once*. This implies that some variables may not be written at all. We derive a dSAC program using the FADA algorithm, therefore, parameters introduced by FADA are present in the dSAC as well. The values of these parameters in dSAC are assigned using control arrays. The generation of the control arrays has been studied in [21], whereas, in this chapter, we present an extension to this procedure. Similar to the SAC, the dSAC can be represented in Polyhedral Reduced Dependence Graph (PRDG) [15] form. In the fourth step, the topology of the corre-

sponding PPN is derived, as well as the code executed in each process. Recall, that the PRDG model still exploits (multi-)dimensional arrays for data communication. However, the target model, Polyhedral Process Networks, requires FIFO channels as communication medium. Therefore in this step, the multi-dimensional memory accesses are converted into managed dataflow over FIFO queues.

In the remaining part of this chapter we describe the four steps in greater detail. We illustrate the proposed solution approach using the example shown in Figure 3.1(a). Additionally, in Section 3.6 we discuss how the buffer sizes are computed in the resulted PPN. In Section 3.7, we present an analysis which estimates the execution overhead introduced in the PPNs derived from programs with dynamic loop bounds. Finally, in Section 3.8 we present the conclusions.

3.2 Step 1 (Dynloop-to-WDP)

Consider the `Dynloop` program in Figure 3.1(a). In this program, the upper bound of the *for*-loop at line 3 is determined by an arbitrary function $f(\ldots)$. The upper bound of the inner loop i may change at every iteration of the outer loop j but cannot be changed on iterations of i. More importantly, the values of the upper bound are unknown at compile-time as they are determined at run-time by $f()$.

In order to be able to apply our solution approach, we assume that the range of the values that function $f()$ may have is finite. This is particularly true for all programs that execute in finite memory, i.e., the programs we are interested in.

Then, without altering the functionality, we modify the initial `Dynloop` program to the program shown in Figure 3.2(a). Such modification is general and applicable to any `Dynloop` program. First, we substitute the upper bound of the loop at line 3 in Figure 3.1(a) with a constant equal to the maximum value of $f()$, denoted by `max_f`, see line 4 in Figure 3.2(a). For example, for the program in Figure 3.2(a), in order for the 5th element of array `y[]` to be read at line 10, the value of `max_f` should be greater than 5. We will use `max_f` ≥ 5 in the rest of the chapter.

In general, the value of `max_f` can be determined in 4 different ways:

1. provided by the application/program developers (e.g., by using `pragmas` in the code);
2. calculated by analyzing the arrays' capacity and indexing functions;
3. deduced by studying the ranges of function $f()$;
4. by taking the maximum of the data type used to declare the loop iterator.

For example, consider method (2) above. Assume that the capacity of the array `y[]` is 100 elements. Then, by taking into account the array indexing function at line 4

in Figure 3.1(a) and that the program is correct, we can calculate that the maximum value of iterator *i* and, consequently the `max_f` equals to 100.

Second, we introduce an array `X[]` used to capture the values of the dynamic upper bound at run-time. That is, the elements of `X[]` are written by function $f()$ at line 3 in Figure 3.2(a), just before the *for*-loop. The same array elements are used in evaluating the `if`-condition at line 5 in Figure 3.2(a), which preserves the original program behavior. This newly created program belongs to the class of the *weakly dynamic programs* (WDPs). Since the loop bounds of the program in Figure 3.2(a) are fixed and known at compile-time, we can apply the FADA algorithm on this program to perform dependence analysis.

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f(...)
4    for i = 1 to max_f,
5      if i <= X[j],
S1:      y[i] = F1()
7      endif
8    endfor
9  endfor

S2:[] = F2( y[5] )
```

(a) Newly created WDP program

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f(...)
4    for i = 1 to max_f,
5      if i <= X[j],
S1:      y_1[j,i] = F1();
7      endif
8    endfor
9   endfor

10 if c1 <= N && c2 == 5,
11   in_0 = y_1[c1,c2]
12 else
13   in_0 = 0
14 endif
S2:[] = F2( in_0 )
```

(b) Initial dSAC

Figure 3.2: A WDP program equivalent to the `Dynloop` program in Figure 3.1(a) and its corresponding dSAC.

The formal description of the FADA algorithm has been given in Section 2.5. In the following section, we demonstrate only the application of FADA on our running example.

3.3 Step 2 (FADA analysis)

The WDP program in Figure 3.2(a) has two statements $S1$ and $S2$ which communicate through array `y[]`. Statement $S2$ is not enclosed in any loops, therefore its iteration vector is empty, i.e., an operation of statement $S2$ is written as $\langle S2, () \rangle$. According to FADA, for pair $S1S2$, we build the system of linear inequalities shown in Table 3.1 which corresponds to Equation 2.9. Constraint (c1) in Table 3.1 describes all possible source iterations of statement $S1$, i.e., its iteration domain. The vector of parameters (α_j, α_i) stores the iteration point (j_1, i_1) of statement $S1$ where writing to array `y[]` may occur.

The system shown in Table 3.1 is used to formulate a PILP problem specified by

$\mathbf{Q}_{S1S2}((\alpha_j, \alpha_i))$	
$1 \leq j_1 \leq N \wedge 1 \leq i_1 \leq \texttt{max\_f}$	(c1)
$j_1 = \alpha_j \wedge i_1 = \alpha_i$	
$i_1 = 5$	(c2)
`true`	(c3)

Table 3.1: An example of system (2.9) for S1S2 pair.

Equation (2.10). After solving the PILP problem, the approximated source operation defined in Equation 2.11 for statement $S2$ is:

$$\sigma(\langle S2, ()\rangle, (\alpha_j, \alpha_i)) = \left| \begin{array}{ll} \textbf{if} & \alpha_j \leq N \wedge \alpha_i = 5 \\ \textbf{then} & \langle S1, (\alpha_j, \alpha_i)\rangle \\ \textbf{else} & \perp . \end{array} \right. \tag{3.1}$$

From Solution 3.1 above, we see that for read operation $\langle S2, ()\rangle$ there is one data source. If, for at least one iteration $(j_1, 5)$ of statement $S1$, the condition at line 5 in Figure 3.2(a) is evaluated to `true`, then the source is statement $S1$. Otherwise, the source for `y[5]` is undefined and statement $S2$ will use the initial value of `y[5]`. For the sake of brevity, the initialization of array `y[]` is omitted in the example.

The graphical representation of Solution 3.1 is illustrated in Figure 3.3. This figure shows the iteration domain (j, i) of statement S1 in one possible instance of the dynamic program shown in Figure 3.2(a). It is assumed that $N = 10$ and $\texttt{max\_f} = 10$. Black dots represent the iterations when statement S1 is executed at run-time, i.e., the `if`-condition at line 5 evaluated to `true`. The vector of parameters (α_j, α_i) points at the last operation of the source statement $\langle S1, (j_1, i_1)\rangle$ which will be needed by the read operation $\langle S2, ()\rangle$. For the example in Figure 3.3, the last writing to `y[5]` occurred when $j = 8$ and $i = 5$. Therefore, $(\alpha_j, \alpha_i) = (8, 5)$.

3.3.1 Initial dSAC

The solution provided by FADA is used to modify the WDP program in order to capture the identified dependencies in an explicit way. The result of the modification for our running example is shown in Figure 3.2(b) which is in a dynamic single-assignment-code (dSAC) form. The dSAC is an extension of the SAC [4]. In contrast to SAC where every variable is written exactly once, in dSAC every variable is written *at most once*. This implies that some of the variables may not be written at all.

Based on Solution 3.1, we modify the WDP in Figure 3.2(a) and generate the dSAC

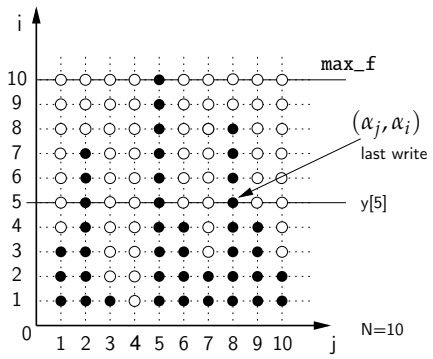


Figure 3.3: Representation of Solution 3.1 for the instance of the program in Figure 3.2(a).

in Figure 3.2(b) by inserting the code lines 10-14 shown in Figure 3.2(b). This code is needed to implement array element accesses such that the dependencies identified by FADA are respected. For example, the `if`-condition at line 10 implements Solution 3.1. Recall that when the `if`-condition evaluates to true, then the source of the data is statement S1. This is captured at line 11. Otherwise, statement S2 will use the initial value of `y[5]`. Assume that in our example, `y[5]` has been initialized to zero. Therefore, at line 13, the input argument for statement S2 has been set to zero as well.

Recall that to deal with a dynamic `if`-condition, for every pair of statements the FADA algorithm introduces vector of parameters that corresponds to the iteration vector. In our example, there are two parameters (see line 10 in Figure 3.2(b)) which are reflected in the following way. Parameter `c1` corresponds to α_j. It is related to iterator j and may have values `c1` $\in [1..\texttt{N}]$. Parameter `c2` corresponds to α_i. It is related to iterator i and may have values `c2` $\in [1..\texttt{max\_f}]$. The meaning of the parameter values in this program is to indicate the last iteration of j when function $F1()$ has been executed at the fifth iteration of i. The values of parameters `c1` and `c2` are unknown at compile-time. They are determined at run-time, during the execution of the program. Therefore, we need a mechanism to generate and propagate the values at run-time in a way that keeps the correct program behavior.

3.4 Step 3 (Control arrays)

In order to keep the functionality of the dSAC equivalent to the functionality of the initial WDP, we introduce *local* and *global* control arrays that are used to initialize and propagate values of parameters introduced by FADA at run-time.

3.4.1 Local control arrays

A *local* control array is added for the set of parameters introduced by FADA and is used to store values of the set of parameters for every iteration. We illustrate the idea of local control arrays on the example in Figure 3.3.

Figure 3.3 depicts the iteration domain (j, i) of statement S1 shown at line 6 in Figure 3.2(a). Black dots are iterations when statement S1 is executed at run-time, i.e., the `if`-condition at line 5 evaluated to `true`. Parameters introduced by FADA in the previous step happen to take up the values of iteration vectors when the last writing needed by a read operation occurred. It is not possible to determine such iterations at compile-time. Therefore, we use a local control array to store the values of all iterations when a source statement is executed (black dots).

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
9    endfor
10 endfor

11 (c1,c2) = lcl_c1c2[5]
12 if c1 <= N && c2 == 5,
13   in_0 = y_1[c1,c2]
14 else
15   in_0 = 0
16 endif
17 [] = F2( in_0 )
```

(a) Initial dSAC shown in Figure 3.2(b) with local control array

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
S1:    ctrl_c1c2[i] = lcl_c1c2[i]
10   endfor
11 endfor

S2:(c1,c2) = ctrl_c1c2[5]
13 if c1 <= N && c2 == 5,
14   in_0 = y_1[c1,c2]
15 else
16   in_0 = 0
17 endif
18 [] = F2( in_0 )
```

(b) Modified dSAC code with new global control array

```
1  parameter N 1 10;

2  for j = 1 to N,
3    X[j] = f()
4    for i = 1 to max_f,
5      if i <= X[j],
6        y_1[j,i] = F1()
7        lcl_c1c2[i] = (j,i)
8      endif
9      ctrl_c1c2_1[j,i] = lcl_c1c2[i]
10   endfor
11 endfor

12 (c1,c2) = ctrl_c1c2_1[N, 5]
13 if c1 <= N && c2 == 5,
14   in_0 = y_1[c1,c2]
15 else
16   in_0 = 0
17 endif
18 [] = F2( in_0 )
```

(c) Final dSAC

Figure 3.4: Examples of the initial dSAC with a local control array, the modified dSAC with a global control array, and the final dSAC.

For our example in Figure 3.2(b), a new local control array of vectors `lcl_c1c2[]` is introduced to the program as shown in Figure 3.4(a). The components of each vector correspond to parameters `c1` and `c2` derived by the FADA analysis for pair S1S2. We use the original index function used with the data variable `y`, i.e., `y[i]`, to perform the access to the local control arrays, i.e., `lcl_c1c2[i]`. In order to distinguish iterations where parameters values have been stored, the elements of the control arrays must be initialized with values that are greater than the maximum value of the corresponding parameters. Recall that for our example, parameter `c1` $\in [1..\mathtt{N}]$ and `c2` $\in [1..\mathtt{max\_f}]$. Therefore, the corresponding local control array is initialized as follows:

$$\forall \mathtt{i} : 1 \leq \mathtt{i} \leq \mathtt{max\_f} : \quad \mathtt{lcl\_c1c2}[\mathtt{i}] = (\mathtt{N}+1, \mathtt{max\_f}+1). \tag{3.2}$$

For the sake of brevity, this initialization is not shown in Figure 3.4(a). Writing to the local control array is performed just after function $F1()$, see line 7 in Figure 3.4(a). This guarantees that when the function is executed, the current iteration vector is stored in the control array.

The values of the local control array are propagated and assigned to the parameters `c1` and `c2` at line 11. These parameters are used to evaluate the conditions at line 12 which determine the source of the data for function $F2()$. With the introduction of the local control array to the program shown in Figure 3.4(a), this program is input-output equivalent to the program in Figure 3.2(a).

3.4.2 Global control arrays

Unfortunately, introducing *local* control arrays to the dSAC code violates the property that "every variable is written *at most once*". For example, local control array `lcl_c1c2[i]` that initializes parameters `c1` and `c2` at line 11 in Figure 3.4(a) is not in a single assignment form, i.e., elements of `lcl_c1c2[i]` may be written more than once (see line 7). Therefore, the program in Figure 3.4(a) is not in a dSAC form. In order to be able to create a process network, as discussed later in Step 4, and most importantly, to create the FIFO channels used for transferring data, the corresponding data variables/arrays must be in a single assignment form. Below, we explain how such control array is transformed into a single assignment form.

In order to represent the program in Figure 3.4(a) as dSAC, we need to identify the relation between writing to and reading from the control array. Thus, we need to perform dataflow analysis for the local control array, where the writings to the control array occur inside a block surrounded by a dynamic `if`-condition. We achieve this in the following way. While keeping the same functionality, we modify the program by introducing and additional *global* control array (`ctrl_c1c2[]`) *outside* the block surrounded by the dynamic `if`-condition, see lines 9 and 12 in Figure 3.4(b). This program is input-output equivalent to the program in Figure 3.4(a). The new control array is written (line 9) at every iteration of the *for*-loops and takes the same values as the local control array `lcl_c1c2[]`. Consequently, we can perform the *static* exact array dataflow analysis (presented in Section 2.4) on control array `ctrl_c1c2[]`. We can always do this, because the introduced new array is not surrounded by the dynamic `if`-condition.

$\mathbf{Q}_{S1S2}()$	
$1 \leq j_1 \leq N \wedge 1 \leq i_1 \leq$ `max_f`	(c1)
$i_1 = 5$	(c2)
`true`	(c3)

Table 3.2: An example of system (2.5) for the control arrays at lines 9 and 12.

For the EADA analysis we need to build a system of linear inequalities as it has been shown in Section 2.4. The system for pair $S1S2$ at lines 9 and 12 from Figure 3.4(b) is built in Table 3.2. Recall, that `max_f` is a scalar and in this example we assume that `max_f` ≥ 5. After finding the maximum of the system according to Equation (2.7), the final solution and the source operation is:

$$\sigma(\langle S2, ()\rangle) = \langle S1, (N, 5)\rangle.$$

Based on this solution, we replace the original one-dimensional array `ctrl_c1c2[]`, see lines 9 and 12 in Figure 3.4(b), with two-dimensional array `ctrl_c1c2_1[,]` shown at lines 9 and 12 in Figure 3.4(c). The program in Figure 3.4(c) is in a dSAC form because the new global control array `ctrl_c1c2_1[]` used to initialize parameters `c1` and `c2` is in a single assignment form. This dSAC is the final input-output equivalent representation of our running example which is the `Dynloop` program in Figure 3.1(a). We use this final dSAC to generate a process network which is explained in the next section.

3.5 Step 4 (PPN generation)

In this step of our solution approach, we describe how the processes and FIFO channels are created from the corresponding final dSAC program. The dSAC specification has an equivalent polyhedral representation called Polyhedral Reduced Dependence Graph (PRDG) [15] form. This representation can be used for code generation for each process [14]. For the illustrative purposes, instead of polyhedral model we will use the dSAC form to demonstrate how processes of a PPN are generated.

Recall that according to Definition 2.3.1, a PPN consists of autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels. A process of a PPN consists of a target *function*, *input ports* and *output ports*. The target function specifies how data tokens from input streams are transformed to data tokens to output streams. The input and output ports are used to connect a process to FIFO channels. Data read from the input ports is used to initialize the function arguments. Data produced as a result of the function execution is written to the output ports. Section 2.1.1 demonstrated how a process can be compactly represented mathematically using the Polyhedral Reduced Dependence Graph (PRDG) [15]. This polyhedral representation is used to generate node domains (see Definition 2.3.2) and input/output port domains (see Definition 2.3.3 and Definition 2.3.4).

The procedure for PPN generation from the final dSAC consists of 3 substeps. First, based on the final dSAC representation of a `Dynloop` program derived in the previous step, the topology of the PPN is created. The topology is created by instantiating processes and communication channels. Second, the internal code structure of each process is derived from the final dSAC specification. It is important to note, that in this substep, the created communication channels are not FIFOs but multi-dimensional arrays. Third, the multi-dimensional arrays that are used for data communication between function statements in the final dSAC are replaced by FIFO

channels. In other words, we replace the multi-dimensional array accesses in the code of each process with a read/write primitives to implement synchronization through blocking read/write on FIFO channels. This substep is called *Linearization*. Below, we explain the three substeps in more detail using the final dSAC in Figure 3.4(c).

3.5.1 Topology creation of a PPN (substep 1)

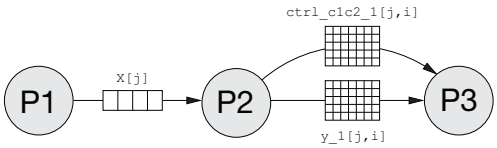


Figure 3.5: The topology of the PPN derived from the dSAC in Figure 3.4(c).

The PPN corresponding to the dSAC in Figure 3.4(c) is shown in Figure 3.5. This PPN consists of 3 processes and 3 communication channels. We explain how these processes and communication channels are created. In our approach, a process is created for every function statement in the dSAC program. Therefore, the PPN in Figure 3.5 has three processes: process $P1$ corresponds to function $f()$ at line 3 in Figure 3.4(c), process $P2$ corresponds to function $F1()$ at line 6, and process $P3$ corresponds to $F2()$ at line 18 in the same figure. The three communication channels correspond to arrays which are in a single assignment form in the dSAC in Figure 3.4(c). These arrays are: one-dimensional array `X[j]` at line 3 and 5 in Figure 3.4(c), two-dimensional data array `y_1[j,i]` at lines 6 and 14, and one two-dimensional control array `ctrl_c1c2_1[j,i]` at lines 9 and 12 in the same figure. Recall that array `X[j]` is in a single assignment form because of the way we introduced this array in Step 1 of our solution approach. Array `y_1[j,i]` is the single assignment form of array `y[i]` derived by applying the FADA analysis on the WDP program in Figure 3.2(a) as described in Step 2 of our solution approach. The control array `ctrl_c1c2_1[j,i]` is introduced and transformed into a single assignment form in Step 3 of our solution approach. In the following substep, we describe how the internal code structure of each process is created.

3.5.2 Internal code structure generation (substep 2)

Consider Figure 3.6 where the internal code structures of processes $P1$, $P2$ and $P3$ of the PPN in Figure 3.5 are shown. Below we explain how these code structures are derived from the corresponding dSAC specification depicted in Figure 3.4(c).

The Node domain of a process introduced by Definition 2.3.2 is the iteration domain of a corresponding statement in the dSAC program. For example, the node domain of process $P2$ is formed by the iteration domain of function $F1$ defined by lines 2, 4, and 5 in Figure 3.4(c). Additionally, the code accessing data and control arrays is

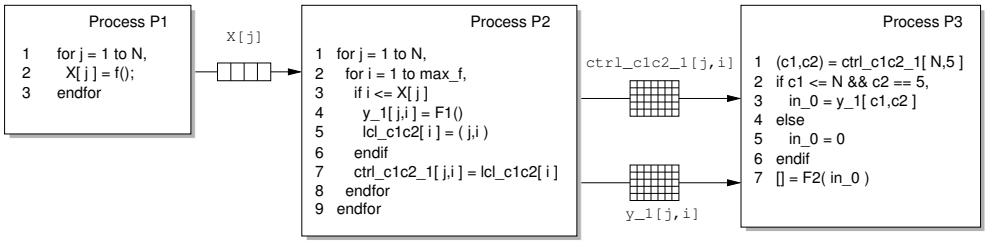


Figure 3.6: The internal code structure of each process in the PPN derived from the dSAC in Figure 3.4(c).

added to the code of a process. For example, lines 6–11 are added to the internal code structure of process *P*2 shown in Figure 3.6. Similarly, the internal code structure of processes P1 and P3 are formed by lines 2–3 and 12–18, respectively, from the dSAC shown in Figure 3.4(c).

3.5.3 Linearization (substep 3)

At this point, the processes of the PPN communicate data via multi-dimensional arrays. In this substep, we explain how the multi-dimensional arrays are replaced with FIFO channels. This process is called *Linearization*.

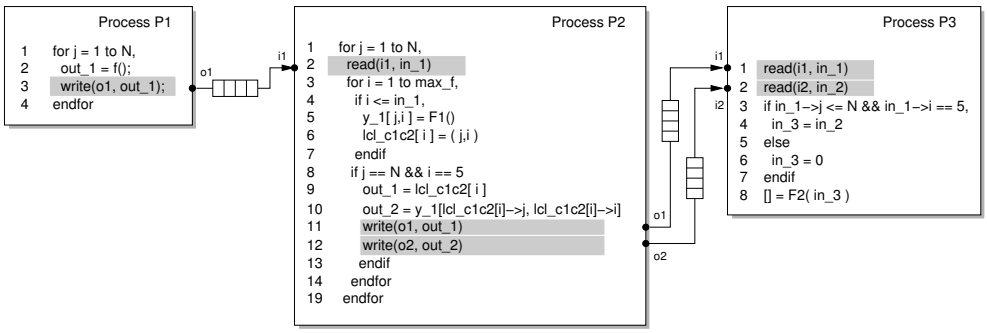


Figure 3.7: The final PPN derived from the program in Figure 3.1(a).

In the PPN depicted in Figure 3.6, processes are connected with communication channels which are the multi-dimensional arrays used in the dSAC shown in Figure 3.4(c). However, as explained in Section 1, the processes in a PPN communicate via FIFOs and synchronize using a blocking read/write on an empty/full FIFO channel, i.e., an execution of a process is suspended if it tries to read from an empty FIFO channel, or tries to write to a full channel, respectively. Therefore, in order to generate a PPN, the multi-dimensional array accesses have to be replaced with corresponding *write* and *read* operations on FIFO channels.

The Linearization is implemented using the approach presented in Chapter 5 of this

dissertation. While there, the approach is discussed in full details, here, we present only the summary of the Linearization approach applied to our running example.

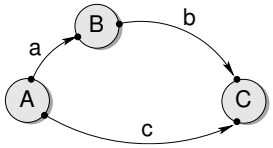
The approach presented in Chapter 5 identifies the communication characteristics of a data exchange in a pair of processes. Based on this information, the multi-dimensional array accesses are replaced with one-dimensional FIFO accesses. The result of the linearization applied on the multi-dimensional arrays in Figure 3.6 is shown in Figure 3.7. In each process, the multi-dimensional arrays accesses are substituted by read/write primitives from/to FIFO channels. Internally, these read-/write primitives realize the blocking synchronization between processes. For example, writing to the global control array at line 7 of process $P2$ in Figure 3.6 is substituted by writing to the FIFO at line 11 in process $P2$ in Figure 3.7.

The communication read/write primitives access the FIFO channels through ports. That is, every process has a set of input ports and a set of output ports connected to FIFO channels. For example, process $P2$ reads from a single channel via port `i1` at line 2 and writes data to two channels via ports `o1` and `o2` at lines 11 and 12, respectively. Additionally, we apply the iteration domain reconstruction of ports described in [14] to avoid transferring more data tokens than needed. For details, we refer to [14].

3.6 Calculation of deadlock-free buffer sizes

```
1   for i = 0 to N,
2     a[i] = A()
3   endfor
4   for j = 1 to N,
5     b[j] = B(a[j-1])
6   endfor
7   for k = 1 to N,
8     C(a[k-1],b[k])
9   endfor
```

(a) Sequential Program



(b) PPN

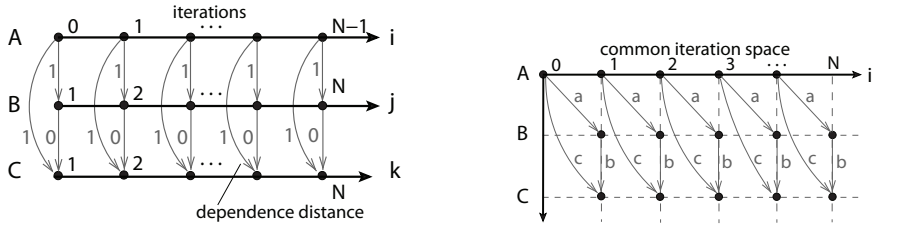
Figure 3.8: An example of a SANLP program and its PPN graph.

Finally, we discuss how we compute the sizes of FIFO channels that guarantee a deadlock-free execution of a PPN derived from a `Dynloop` program. First, we explain the procedure for computing buffer sizes in a PPN derived from a static affine nested loop program (SANLP). Then, we explain how to use this procedure to compute buffer sizes for a PPN derived from a `Dynloop` program.

Computing minimal deadlock-free buffer sizes is a non-trivial global optimization problem. This problem becomes easier if we first compute a deadlock-free schedule of the PPN and then compute the buffer sizes for each channel individually. Note that this schedule is only computed for the purpose of computing the buffer sizes and is discarded afterwards because the processes in our PPNs are self-scheduled due to the blocking read/write synchronization mechanism. Although the schedule

we compute may not be optimal, our computations do ensure that a valid schedule exists for the computed buffer sizes. The schedule is computed using a greedy approach. This approach may not work for process networks in general, but it does work for PPNs derived from static affine nested loop programs.

The basic idea is to place all iteration domains in a common iteration space at an offset such that the dependences in the initial program are respected. The offset is computed by the scheduling algorithm described in [49]. By fixing the offsets of the iteration domain in the common space, we have therefore fixed the relative order between any pair of iterations from any pair of iteration domains. The algorithm starts by computing for any pair of connected processes, the minimal dependence distance vector, being the difference between a read operation and the corresponding write operation. Then, the processes are greedily combined, ensuring that all minimal distance vectors are (lexicographically) positive. The end result is a schedule that ensures that every data element is written before it is read. For more information on this algorithm, we refer to [49], where it is applied to perform loop fusion on SANLPs.



(a) Representation of data dependencies

(b) Schedule for minimum buffer sizes

Figure 3.9: Representation of the data dependencies between statements on the program in Figure 3.8(a), and the global schedule computed for the same program for minimum buffer sizes.

As an example, consider the sequential program shown in Figure 3.8(a). It results in the process network in Figure 3.8(b). The data dependencies are depicted in Figure 3.9(a). The horizontal axes illustrate the single dimension of the iteration domains of the processes (function calls) A, B and C, and the arrows show the data dependencies. The value of the dependence distances are shown next to each arrow. As a next step, a valid global schedule is computed by placing (offsetting) processes together in a way that keeps the distance between write operations and the corresponding read operations minimal.

The result is shown in Figure 3.9(b). In this figure, next to each arrow, we also depict the names of the FIFO channels used to propagate the corresponding data at each iteration, e.g., FIFO a is used to propagate data between processes A and B. In the common iteration space, the horizontal axis represents the single dimension of the problem and the vertical axis represents the additional dimension that orders the statements lexicographically.

To compute the buffer sizes for each FIFO, we compute the number of read iterations $R(i)$ that are executed before a given read operation i and subtract the resulting expression from the number of write iterations $W(i)$ that are executed before the given read operation:

$$\#elements\ in\ FIFO\ at\ operation\ i:\ W(i) - R(i)$$

This computation can be performed entirely symbolically using the `barvinok` library [50] that efficiently computes the number of integer points in a parametric polytope. The result is a piecewise (quasi-)polynomial in the read iterators and the parameters. Then, the required buffer size is the maximum of this expression over all read iterations:

$$FIFO\ size\ =\ \max_{i}(\ W(i) - R(i)\ )$$

To compute the maximum symbolically, we apply the Bernstein expansion [51] to obtain a parametric *upper bound* on the expression.

Below, we show how the buffer sizes are computed based on the schedule in Figure 3.9(b). Consider FIFO a. Let the number of elements written to the FIFO by process A before iteration i is denoted as $W_A^a(i)$ and the number of elements read from the same FIFO by process B before iteration i is denoted as $R_B^a(i)$. Then, for every iteration $i, i \in [1, N]$, we compute the difference $W_A^a(i) - R_B^a(i)$ and assign the maximum difference as the buffer size of FIFO channel a. For example, consider the fourth iteration of the common iteration spaces ($i = 3$). Then:

$$\begin{aligned} W_A^a(3) &= 3, \\ R_B^a(3) &= 2, \\ W_A^a(3) - R_B^a(3) &= 3 - 2 = 1. \end{aligned}$$

Due to the uniform data dependences in the example, $W_A^a(i) - R_B^a(i) = 1, \forall i \in [1, N]$ and consequently the size of FIFO channel $a = max(\ W_A^a(i) - R_B^a(i)\ ) = 1$. In the same way, we compute the buffer sizes of the remaining FIFOs, i.e.,

$$\begin{aligned} &\text{size of FIFO channel } b = max(\ W_B^b(i) - R_C^b(i)\ ) = 0, \\ &\text{size of FIFO channel } d = max(\ W_A^c(i) - R_C^c(i)\ ) = 1. \end{aligned}$$

If some of the computed FIFO buffer sizes equal to zero, then size 1 is assigned to all such FIFO channels.

In contrast to PPNs derived from SANLPs, the PPNs derived from `Dynloops` contain two types of channels: control and data FIFO channels. Control channels realize dependencies between global control arrays presented in Step 3 (see Section 3.4.2). These dependencies are defined by the static part of a `Dynloop` program. Therefore, for control channels we can apply the procedure for computing buffer sizes described above. For example, the control arrays at lines 9 and 12 in Figure 3.4(c) are global and we can use the method described above to compute buffer sizes.

Data channels realize data dependencies between function statements of a `Dynloop` program. In contrast to SANLP programs, in `Dynloop` programs some statements are guarded by dynamic `if`-conditions. Consequently, the iteration domains of these statements as well as the rate and the exact amount of data tokens that will be transferred over the corresponding data channels are unknown at compile-time. Therefore, we cannot use directly the method described above to compute buffer sizes. To be able to handle the dynamism of `Dynloop` programs we have to follow a conservative strategy, i.e., we have to calculate buffer sizes such that to provide enough space to run any possible instances of the dynamic program. There is always one instance of a dynamic program that requires the largest buffer sizes. It is the instance when the iteration domains of input/output ports of all FIFO channels are the largest. These iteration domains are the largest when the dynamic `if`-condition that determine these domains evaluate to `true`. In our procedure for calculating FIFO buffer sizes in channels derived from a `Dynloop` program, we modify the iteration domains of input/output ports of all FIFO channels, such that all dynamic `if`-conditions defining any of these iteration domains evaluate always to `true`. This means in practice, that we ignore/remove the dynamic `if`-conditions from the FIFO calculation. Therefore, again we can apply the procedure described above to the resulted channels.

3.7 Overhead Analysis

In this section, we discuss the overhead in the generated process networks, which results from the proposed approach for systematic parallelization of sequential programs with dynamic loop bounds. There are two types of overhead in the generated process networks, i.e., memory and execution time overhead. The memory overhead is due to the introduced control arrays, as well as, the created dataflow and control FIFO channels. It highly depends on the characteristics of the application being parallelized (see Section 6.1, Memory overhead and Table 6.2). Therefore, it is very difficult to be analyzed systematically. However, we can systematically analyze the execution time overhead which is introduced by the approach we propose in this chapter. This overhead is caused by the execution of some 'dummy' iterations not present in the initial sequential program. Below, we discuss this overhead in details. Recall that in our approach, we substitute a dynamic upper loop bound with the maximum value (max_f) that the bound may have during the execution of the program. Then at run-time, the actual number of iterations at which a function executes is determined by the behavior of the application and the current value of the dynamic loop bound. This means that if the actual number of executions (x) is smaller than the maximum number, then the corresponding process performs ($max\_f - x$) 'dummy' iterations. The overhead, we consider, is the time spent in performing these "dummy" iterations.

It is important to note that it is difficult to determine the exact amount of the overhead because it depends on values which are determined and change at run-time. Below, we define the overhead and determine how it varies for particular range of

its terms. Assume that max_f is the maximum value of a dynamic loop bound and x represents the actual number of iterations in which a process executes its associated function. When a function executes, it takes W_x time units. Performing a 'dummy' iteration takes W time units, respectively. This is the time spent in one iteration but not executing the corresponding function. Then, for any given values of max_f, x, W_x, and W, the total execution time (T_{ex}) is:

$$T_{ex} = x(W_x + W) + (max\_f - x)W, \tag{3.3}$$

where $x(W_x + W)$ is the time spent on real computation (T_{real}) and $(max\_f - x)W$ is the extra time spent performing 'dummy' iterations. Consequently, we can compute the introduced execution overhead as follows:

$$\frac{T_{ex}}{T_{real}} = \frac{x(W_x + W) + (max\_f - x)W}{x(W_x + W)} = 1 + \frac{(max\_f - x)W}{x(W_x + W)},$$

where the percentage of the execution overhead ($Ovhd$) is:

$$Ovhd = \frac{(max\_f - x)W}{x(W_x + W)} \cdot 100 = \frac{(max\_f - x)}{x} \cdot \frac{W}{(W_x + W)} \cdot 100\ [\%] \tag{3.4}$$

Equation 3.4 shows that the overhead depends on two ratios. The first one, $\frac{(max\_f - x)}{x}$, depends on *i)* the application characteristics, which determine max_f, and *ii)* the execution behavior, which determines the values of x at run-time. The second ratio is related to the computation performed by a process (executed on a particular processor) as it represents the ratio between the time to perform a 'dummy' iteration and the time spent on actual computing. Figure 3.10 illustrates the amount of overhead for the following ranges of the two ratios in Equation 3.4:

1. $0 \leq \frac{max\_f - x}{x} \leq 2 \Rightarrow$ for any value of max_f, $\frac{max\_f}{3} \leq x \leq max\_f$;
2. $0.01 \leq \frac{W}{W_x + W} \leq 0.5 \Rightarrow$ for any value of W, $W \leq W_x \leq 99 \cdot W$.

These ranges capture the characteristics for a wide spectrum of applications and their behavior. Moreover, our experience shows that if a particular application has sufficient inherited parallelism, then the approach we propose to parallelize sequential programs with dynamic loop bounds can lead to performance speed-up if the two ratios stay within the specified ranges above.

In case $x = max\_f$, there is no overhead (see the right part of Figure 3.10) because there are no 'dummy' iterations to be executed ($max\_f - x = 0$). Then, by decreasing the value of x, the overhead increases. The rate of the increase is determined also by the value of $\frac{W}{W_x + W}$. The values of this ratio used in the figure capture functions

with low and high workload. The lowest workload we consider is $W_x = W$, i.e., the time to execute the corresponding function is equal to the time of a 'dummy' iteration (see the back plane of the figure). We use such a low workload to illustrate some extreme values of the overhead. For example, when $W_x = W$ and $x = max\_f/2$ the maximum overhead is 50%. The combined effect of both ratios leads to 100% overhead when $W_x = W$ and $x = max\_f/3$, see the left part of Figure 3.10. In contrast, functions with high workload, i.e., $50 \cdot W \leq W_x \leq 99 \cdot W$, lead to very low overhead. For example, even if $x = max\_f/3$, the introduced overhead is around 5-10% as it can be observed at the bottom-left part in the figure. This indicates that the approach we propose is not sensitive to functions with high workloads.

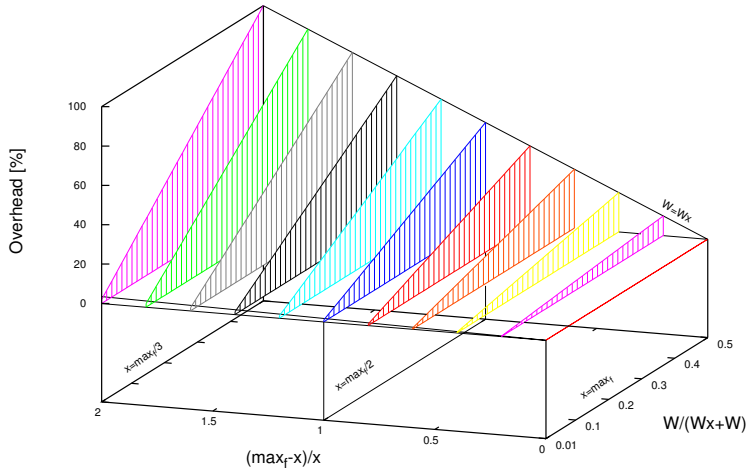


Figure 3.10: The amount of introduced overhead.

For easier evaluation of the overhead values, we plot the percentage overhead as a color map in Figure 3.11. From this figure, it is seen that overhead above 35% is present only in $1/4$ of the cases. In addition, $1/16$ of the cases, see the area with overhead $\geq$ 80%, correspond to functions with very low workload and a large number of 'dummy' iterations. For the other $3/4$ of the cases, we would like to emphasize on the following two areas. First, if the ratio $\frac{max\_f - x}{x}$ is smaller than 0.25, then the granularity of the executed functions does not affect the overhead, which is below 10%, see the dark vertical strip on the left part of the figure. This indicates also that for light-weight functions, the overhead will be small if the executed iterations are close to the max_f value. Similarly, in case of functions with high workload ($50 \cdot W \leq W_x \leq 99 \cdot W$), the number of 'dummy' iterations that are executed does not affect the overhead, which again is below 10% – see the dark horizontal strip at

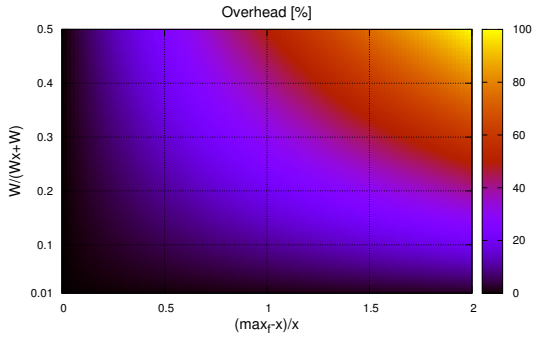


Figure 3.11: Overhead's color map.

the bottom of Figure 3.11. The second area covers almost half of the plot, see the arc-shape stripe in the middle of Figure 3.11. This area shows that even with a large variety of the values of both ratios, the overhead is kept below 35%, which is relatively low. This area also shows that such overhead can be achieved even if one of the ratios goes to its extreme value. For example, 35% overhead is achieved if $\frac{W}{W_x+W}$ reaches its maximum value of 0.5 and $\frac{max\_f-x}{x} = 0.7$.

3.8 Discussion and Summary

In this chapter, we presented a first approach for automated translation of affine nested loops programs with dynamic loop bounds (`Dynloop`) into input-output equivalent polyhedral process networks (PPNs). The problem of deriving a Process Network specification from a sequential program in a systematic and automated way has been addressed by many researchers. The work in [14, 30, 52] reports techniques for automated derivation of Kahn Process Networks (KPNs) [7] from applications specified as *static* affine nested loop programs (SANLPs). The main property of such programs is that everything about the program execution is known at compile-time. However, the *static* restriction limits the applicability of these approaches, i.e., these approaches cannot be applied to the applications that have adaptive and dynamic behavior, such as multimedia applications (MPEG coders/decoders, Smart Cameras, Software Radio), adaptive filters, iterative algorithms, etc. If some of the static restrictions of the SANLPs could be relaxed while keeping the ability to derive PPNs in an automated way, this would significantly extend the range of applications that can be parallelized in an automated way. This inspired the work in [21] where an approach for KPN derivation from Weakly Dynamic Programs (WDP) has been developed. WDPs are more relaxed than the SANLP class of applications where `if`-conditions might be dependent on some information that is unknown at compile-

time and may change at run-time. In this chapter, we further extended the class of applications to `Dynloop` programs from which the PPN specification can be derived in an automated way.

Although, the execution of a `Dynloop` program is not known completely at compile time, we have shown in this chapter that still a `Dynloop` program can be analyzed and transformed into a PPN in a formal, systematic and structured way. To do this, we demonstrated how a `Dynloop` program can be formally represented as a WDP, we employed the Fuzzy Array Dataflow Analysis (FADA) technique, the dynamic Single Assignment Code form and demonstrated how to set the values of parameters introduced by FADA.

In a PPN derived from a `Dynloop` program we distinguish two types of communication FIFO channels depending on the purpose of the communicated data: 1) *data FIFO channels* where computational data used/generated by function calls (tasks) executed inside processes is communicated; 2) *control global FIFO channels* where data that controls the internal sequential behavior of processes is communicated. By sequential behavior of a process we mean the sequential order of execution of function calls inside the process.

The control FIFO channels appear in a PPN derived from a `Dynloop` program because the behavior of `Dynloop` is not know completely at compile-time. The unknown behavior has to be resolved at run-time in the PPN and the control FIFO channels are used to communicate the necessary data to do this. Control FIFO channels do not appear in case a PPN is derived from a static program. This means that the presence of control FIFO channels introduces extra workload and communication overhead that are the consequences of the dynamic nature of the initial application.

Most of the methods and techniques of our approach presented in this chapter have been prototyped in the *pn* [48] compiler and tested on a small set of `Dynloop` programs. Besides this small set and the running example from this chapter, the approach and the prototype software have been applied and validated successfully on a real-life application called Low Speed Obstacle Detection (LSOD). The analysis of a PPN derivation from this application is presented in Chapter 6.

The approach presented in this chapter includes only basic techniques that we have developed in order to derive a PPN automatically from a `Dynloop` program. The results we have obtained from the LSOD application indicated that as a future work some optimization techniques have to be added to the approach that will help improving the quality of the generated PPNs in terms of optimal partitioning of the computation and communication workloads of a `Dynloop` program over processes and channels in the PPN.

Chapter 4

Automated Generation of Polyhedral Process Networks from Affine Nested-Loop Programs with While-loops

In this chapter, we present a first approach for automated translation of affine nested loop programs which contain relaxation II, i.e., *while*-loops (WLAP), into input-output equivalent Polyhedral Process Networks (PPNs). We developed this approach in order to further extend the range of applications that can be parallelized in an automated way. This approach can be automated and implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential programs.

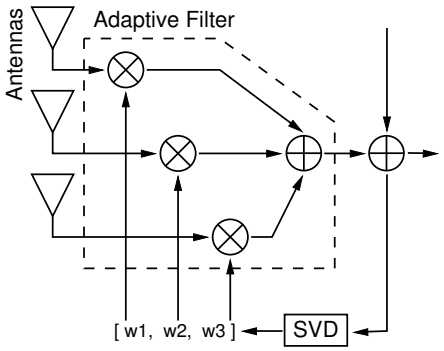
Recall, that in Section 1.1 we briefly introduced the main steps needed to translate a *static* sequential application into a PPN. Additionally, in Section 1.3 we showed that this approach cannot be used on dynamic applications. In this chapter we develop a new approach elaborating in more detail on the new models and techniques that are used in parallelization of programs containing *while*-loops.

The rest of this chapter is organized as follows. In Section 4.1, we present a real-life application that requires while-loop for specification. Further, starting with Section 4.2 until Section 4.6, we present the approach for translation WLAP programs into equivalent PPNs in more detail elaborating on the new models and techniques that are used in parallelization. Finally, in Section 4.7, the conclusions are presented.

4.1 Motivating example

As a motivating example, we use a real-life application from the signal processing domain called Adaptive Beamforming (AB) [25]. With the description of the AB application below, we present a program that has the specific dynamic behavior we consider in this chapter, and we outline the problems introduced by this behavior.

Adaptive Beamforming is a signal processing technique which performs adaptive spatial signal processing with an array of antennas in order to transmit or receive signals in different directions without having to mechanically steer the array. The main property of the AB is the ability to adjust its performance to match the changing signal parameters. Figure 4.1(a) illustrates the AB application. Signals from three antennas are constantly fed into an adaptive filter where they are processed together with adaptive coefficients (ACs) `w1-w3`. ACs are needed to adjust the signals and are recalculated for new signals received from the antennas. This property makes the AB application to be widely used in communications to point an antenna at the changing signal source to reduce interference and improve communication quality. That is why the AB is an important part of modern wireless communication standards, such as IEEE 802.11n (Wifi), 4G, WiMAX, etc.



(a) Adaptive beamforming application

```
1  M = HouseHolder( M )
2  while ( F(M) ),
3    M = QR( M )
4  endwhile
```

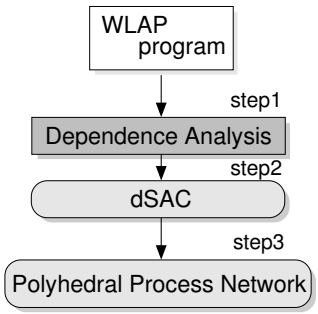
(b) An example of a WLAP program: the SVD algorithm

Figure 4.1: Adaptive Beamforming and the SVD [53] algorithm.

The most computationally intensive part of the AB application is the Singular Value Decomposition (SVD) algorithm. The SVD algorithm performs a factorization of a matrix and is used to produce ACs for the adaptive filter shown in Figure 4.1(a). Pseudo-code of the SVD algorithm is illustrated in Figure 4.1(b). First, a matrix is reduced to a bidiagonal form by the Householder transformation at line 1, and then, the result is diagonalized using an iterative QR algorithm at line 3. Iterative QR is an eigenvalue algorithm, and it is an example of a program which has dynamic control. The program requires a while-loop at line 2 in Figure 4.1(b), as calculated values iteratively converge to eigenvalues until desired precision determined

by function `F()` is achieved. The number of iterations to converge is unknown at compile-time. Since the SVD algorithm cannot be specified as a static program or a program with dynamic `if`-conditions considered in [21] or for-loops with dynamic bounds considered in [26,27] and Chapter 3, the *pn* compiler [3] as well as techniques from [21, 26, 27] and Chapter 3 are unable to handle the program in Figure 4.1(b). Therefore, in this chapter, we propose a solution approach to this problem by introducing a novel procedure for automated translation of affine nested loops programs with while-loops (WLAP) (see Definition 2.2.4) into input-output equivalent PPNs.

Handling the dynamic behavior of while-loops is more difficult compared to dynamic *if*-conditions [21] and for-loops with dynamic bounds (Chapter 3). A for-loop with dynamic loop bounds can be replaced by dynamic `if`-condition with some modifications as it has been shown in [26, 27] and Chapter 3. However, a while-loop cannot be replaced by a for-loop with dynamic bounds. Information about the number of iterations of a while-loop is unknown until the loop has been finished. Whereas the number of iterations of a for-loop with dynamic bounds is known just before the loop starts to execute. This absence of information in a while-loop requires much more advanced analysis compared to analysis of for-loops. In this chapter, we demonstrate the analysis of while-loops in order to translate WLAPs into input-output equivalent PPNs.



(a) Parallelization approach

```
1  parameter EPS 0.005

2  for i = 1 to N,
S1:  y[i] = F1()
S2:  x = F2( y[i] )
W:   while ( x >= EPS )
S3:    x = F3()
7      for j = i+1 to N+1,
S4:      y[j] = F4( y[j-1] )
S5:      x = F5( x, y[j] )
10     endfor
S6:    y[i] = F6( x )
12   endwhile
S7:  out = F7( x )
14 endfor
```

(b) A complex example of a WLAP program.

Figure 4.2: An approach that translates WLAP program into PPNs and a complex example of a WLAP program.

4.2 Solution Overview

The high-level overview of the approach is illustrated in Figure 4.2(a). It starts with an application written as a sequential program that has while-loops similar to one depicted in Figure 1.5(c). First, we find all data-dependency relations in the initial WLAP program by applying the Fuzzy Array Dependence Analysis (FADA) [37, 38]

on it. This analysis, described in Section 2.5, helps to extract the dependent memory accesses and represent an initial program in a form where data dependencies are made explicit. In Section 1.3 we have shown that in a WLAP program *exact* data dependency patterns are unknown at compile time. The FADA analysis allows to parameterize (or approximate) such data dependency patterns with parameters which values are determined at run-time. Second, based on the results of the analysis, we transform the initial WLAP program into a *dynamic Single Assignment Code* (dSAC) representation. dSAC was proposed in [21] as an extension of the SAC [4]. A dSAC program is input-output equivalent to the initial program and it has the property that every variable is written *at most once*. This implies that some variables may not be written at all. We derive the dSAC program using the FADA algorithm, therefore, parameters introduced by FADA are present in the dSAC as well. The values of these parameters in dSAC are assigned using control variables. The generation of control variables constitutes the third step of our solution approach. Control variables have been studied in [21] for programs containing dynamic *if*-conditions, whereas, in this chapter, we present an extension to these procedures which can be applied on WLAP programs. In the last fourth step, the topology of the corresponding PPN is derived, as well as the code executed in each process. In the remaining part of this chapter, we describe the four steps in more detail and we also illustrate our solution approach with the example shown in Figure 4.2(b).

$\mathbf{Q}_{S2S7}(i_7)$	$\mathbf{Q}_{S3S7}(i_7,\alpha,\beta)$	$\mathbf{Q}_{S5S7}(i_7,\alpha,\beta)$	
$1 \le i_2 \le N$	$1 \le i_3 \le N \wedge$ $i_3 = \alpha, 1 \le w_3 \le \beta$	$1 \le i_5 \le N \wedge$ $i_5 = \alpha, 1 \le w_5 \le \beta$ $i_5 + 1 \le j_5 \le N+1$	(c1)
—	—	—	(c2)
$\langle S2,(i_2)\rangle \prec \langle S7,(i_7)\rangle$	$\langle S3,(i_3,w_3)\rangle \prec \langle S7,(i_7)\rangle$	$\langle S5,(i_5,w_5,j_5)\rangle \prec \langle S7,(i_7)\rangle$	(c3)
$\langle S2,(i_7)\rangle$	**if** $\beta \ge 1 \wedge 1 \le \alpha \le i_7$ **then** $\langle S3,(\alpha,\beta)\rangle$ **else** $\perp$.	**if** $\beta \ge 1 \wedge 1 \le \alpha \le i_7$ **then** $\langle S5,(\alpha,\beta,N+1)\rangle$ **else** $\perp$.	SOLUTIONS

Table 4.1: Systems of linear inequalities (2.13) for pairs *S2S7*, *S3S7* and *S5S7* in the program in Figure 4.2(b).

4.3 Step 1 (FADA analysis)

The formal description of the FADA algorithm has been given in Section 2.5. In this step of our solution approach, we demonstrate the application of the FADA analysis on our running example in Figure 4.2(b).

Consider the WLAP program in Figure 4.2(b). An application of the FADA analysis on this program finds all data dependencies between all functional statements communicating data via array `y[]` and scalar `x`. We demonstrate in detail the application of the FADA analysis in order to find source operations for scalar `x` read in statement $S7$. For the other statements, we present the final solutions only and discuss some important observations.

In order to be able to apply the FADA analysis to the program in Figure 4.2(b), we have to capture all iterations of the while-loop at line 5 in an explicit way. We associate an integer iterator `w` with this while-loop. Later, we demonstrate the realization of this iterator in the code.

The candidate source operations for statement $S7$ are in statements $S2$, $S3$ and $S5$. Therefore, in order to find the source operation for statement $S7$ we need to apply the FADA algorithm presented in Section 2.5 on pairs $S2S7$, $S3S7$ and $S5S7$. According to FADA, for all these pairs we build the systems of linear inequalities shown in Table 4.1 which correspond to Equation 2.13. Constraint $c1$ in Table 4.1 describes all possible source iterations of statements $S2$, $S3$ and $S5$. Constraint $c2$ is not stated as data is communicated via scalar `x`. Parameters (α, β), store the iteration point (i_5, w_5) of statement $S5$ and iteration point (i_3, w_3) of statement $S3$ when writing to scalar `x` may occur.

Solutions to the three parametric integer linear problems stated in Table 4.1 are shown in the last row of Table 4.1. For example, in pair $S5S7$ the source operation for `x` is statement $S5$ if condition $\beta \geq 1 \wedge 1 \leq \alpha \leq i_7$ evaluates to true. Otherwise, the source for `x` is not statement $S5$ which is designated by $\perp$. In this case, statement $S7$ will use either the value of `x` assigned somewhere else in the code, or the initial value of `x`.

Finally, after combining the three solutions in Table 4.1, the approximated source operation defined in Equation 2.15 for scalar `x` read in statement $S7$ is:

$$\sigma_{\mathtt{x}}(\langle S7, (i_7, \alpha, \beta)\rangle) = \left| \begin{array}{ll} \textbf{if} & (\beta \geq 1 \wedge 1 \leq \alpha \leq i_7) \\ \textbf{then} & \langle S5, (\alpha, \beta, N+1)\rangle \\ \textbf{else} & \langle S2, i_7\rangle \end{array} \right. \tag{4.1}$$

From Solution 4.1 above, we see that for read operation $\langle S7, (i_7, \alpha, \beta)\rangle$ there are two possible source operations. Depending on the values of the parameter vector (α, β), the source operation is either in statement $S2$ or in statement $S5$. The values of the parameter vector will be determined at run-time.

Similarly, we find the source operations for the other statements. Figure 4.3 shows the source σ functions only for statements $S4$, $S5$, $S6$ and W that include non-trivial dependencies that exist in the program in Figure 4.2(b).

$$\sigma_{\mathtt{y}}(\langle S4,(i_4,w_4,j_4)\rangle) = \left|\begin{array}{ll} \textbf{if} & (j_4 = i_4+1) \\ \textbf{then} & \left|\begin{array}{ll} \textbf{if} & (w_4 = 1) \\ \textbf{then} & \langle S1, i_4\rangle \\ \textbf{else} & \langle S6,(i_4,w_4-1)\rangle \end{array}\right. \\ \textbf{else} & \langle S4,(i_4,w_4,j_4-1)\rangle \end{array}\right. \tag{4.2}$$

$$\sigma_{\mathtt{x}}(\langle S5,(i_5,w_5,j_5)\rangle) = \left|\begin{array}{ll} \textbf{if} & (j_5 = i_5+1) \\ \textbf{then} & \left|\begin{array}{ll} \textbf{if} & (w_5 = 1) \\ \textbf{then} & \langle S3,(i_5,w_5)\rangle \\ \textbf{else} & \langle S5,(i_5,w_5-1,N+1)\rangle \end{array}\right. \\ \textbf{else} & \langle S5,(i_5,w_5,j_5-1)\rangle \end{array}\right. \tag{4.3}$$

$$\sigma_{\mathtt{x}}(\langle S6,(i_6,w_6)\rangle) = \langle S5,(i_6,w_6,N+1)\rangle \tag{4.4}$$

$$\sigma_{\mathtt{x}}(\langle W,(i_W,w_W)\rangle) = \left|\begin{array}{ll} \textbf{if} & (w_W == 1) \\ \textbf{then} & \langle S2, i_W\rangle \\ \textbf{else} & \langle S5,(i_W,w_W-1,N+1)\rangle \end{array}\right. \tag{4.5}$$

Figure 4.3: Source operations for statements $S4$,$S5$,$S6$ and W of the WLAP program in Figure 4.2(b).

4.4 Step 2 (Initial dSAC)

The solutions provided by FADA are used to transform the initial WLAP program in order to expose the identified dependencies in an explicit way. The transformed program shown in Figure 4.4(a) is in dynamic Single Assignment Code (dSAC) form. The dSAC is an extension of the SAC introduced in [4]. In contrast to SAC where every variable is written exactly once, in dSAC every variable is written *at most once*. This implies that some of the variables may not be written at all.

Based on the solutions in the previous step, we transform the initial WLAP program in Figure 4.2(b) and generate the dSAC in Figure 4.4(a) by inserting the highlighted (bolded) code lines into the initial WLAP program. The inserted code is needed to implement array element accesses such that the data dependences in the initial program are respected. The Right-Hand Side (RHS) of code lines 7,11,13,17 and 20

```
1  #parameter EPS 0.005
2  w = 0

3  for i = 1 to N,
S1:  y_1[i] = F1()
5    in_2 = y_1[i]
S2:  x_2[i] = F2( in_2 )
W:   while (in_w = σx(⟨W,(i,w)⟩) >= EPS),
8      w = w + 1
S3:    x_3[i,w] = F3()
10     for j = i+1 to N+1,
11       in_4 = σy(⟨S4,(i,w,j)⟩)
S4:      y_4[i,w,j] = F4( in_4 )
13       in_5_x = σx(⟨S5,(i,w,j)⟩)
14       in_5_y = y_4[i,w,j]
S5:      x_5[i,w,j] = F5( in_5_x, in_5_y )

16     endfor
17     in_6 = σx(⟨S6,(i,w)⟩)
S6:    y_6[i,w] = F6( in_6 )
19   endwhile


20   in_7 = σx(⟨S7,(i,α,β)⟩)
S7:  out = F7( in_7 )
22 endfor
```

(a) Initial dSAC

```
1  #parameter EPS 0.005
2  w = 0
3  ctrl_x_5 = (N+1,0)
4  for i = 1 to N,
S1:  y_1[i] = F1()
6    in_2 = y_1[i]
S2:  x_2[i] = F2( in_2 )
W:   while (in_w = σx(⟨W,(i,w)⟩) >= EPS),
9      w = w + 1
S3:    x_3[i,w] = F3()
11     for j = i+1 to N+1,
12       in_4 = σy(⟨S4,(i,w,j)⟩)
S4:      y_4[i,w,j] = F4( in_4 )
14       in_5_x = σx(⟨S5,(i,w,j)⟩)
15       in_5_y = y_4[i,w,j]
S5:      x_5[i,w,j] = F5( in_5_x, in_5_y )
17       ctrl_x_5 = (i,w)
18     endfor
19     in_6 = σx(⟨S6,(i,w)⟩)
S6:    y_6[i,w] = F6( in_6 )
21   endwhile


22   (α,β) = ctrl_x_5
23    in_7 = σx(⟨S7,(i,α,β)⟩)
S7:  out = F7( in_7 )
25 endfor
```

(b) Modified dSAC with control variable

Figure 4.4: Examples of the initial dSAC and the modified dSAC with control variables.

implement the source σ functions depicted in Solution 4.1 and in Figure 4.3 found by FADA in the previous step of our solution approach. These source σ functions should be interpreted as code lines determined by Solution 4.1 and the solutions in Figure 4.3. For example, variable `in_5_x` at line 13 in Figure 4.4(a) is assigned by the source σ_x function defined by Solution 4.3 in Figure 4.3. This solution finds a source for scalar `x` read in statement $S5$ at line 9 in Figure 4.2(b). The whole line 13 in Figure 4.4(a) should be interpreted as the code in Figure 4.5. The code represents the σ_x function defined by Solution 4.3. Similarly, the other σ functions are represented in the code of dSAC.

```
if (j == i+1),
  if (w == 1),
    in_5_x = x_3[i,w]
  else
    in_5_x = x_5[i,w-1,N+1]
  endif
else
  in_5_x = x_5[i,w,j-1]
endif
```

Figure 4.5: An interpretation of σ_x function for statement $S5$.

Additionally, we transform the while-loop at line 5 in the initial program in Fig-

ure 4.2(b) in order to implement data dependency relations for the while-loop's condition. First, we introduce the iterator `w` in order to capture all iterations of the while-loop. This iterator is initialized at line 2 and explicitly incremented at line 8 in Figure 4.4(a). Second, we replace line 5 in the initial program in Figure 4.2(b) with line 7 in Figure 4.4(a) implementing the same condition function. The source σ_x function defined by Solution 4.5 in Figure 4.3 should be interpreted in the same way as explained above.

Recall that to deal with a *while*-loop, the FADA algorithm introduces a vector of parameters to the solutions. In our example, a vector of parameters (α, β) is introduced at line 20 in Figure 4.4(a) by Solution 4.1. At this line, a source operation for scalar `x` read in RHS of statement $S7$ is determined. Solution 4.1 is approximate, as the potential source statement $S5$ is inside the while-loop. Parameter α is related to iterator i and takes values $\alpha \in [1..\mathtt{N}]$. Parameter β is related to iterator w and takes values $\beta \geq 1$. The meaning of the parameter vector values in this program is to indicate the last iteration (i, w) when statement $S5$ has been executed. The values of parameters α and β are determined at run-time, during program execution. Therefore, we need a mechanism to generate and propagate the values of parameters at run-time in a way that keeps the correct program behavior.

4.5 Step 3 (Control variables)

In order to keep the functionality of the dSAC equivalent to the functionality of the initial dynamic program with *while*-loops, we introduce control variables used to propagate parameter values at run-time. That is, an array of control variables is added for every parameter vector introduced by FADA. A control variable is used to store a parameter vector value for every iteration. For our running example, a new control variable `ctrl_x_5` is introduced at lines 3, 17 and 22 in the program shown in Figure 4.4(b). It stores parameter vector (α, β), derived by FADA in Step 1 of our solution approach. To access a control variable, we use *the same* indexing function as in the corresponding data array. In our example, the new control variable `ctrl_x_5` is a scalar, as it corresponds to the data scalar `x`.

The control variables must be initialized with values that are never taken by the corresponding parameters. Recall that for our example, parameter $\alpha \in [1..\mathtt{N}]$ and $\beta \geq 1$. Therefore, the corresponding control variable `ctrl_x_5` is initialized at line 3 in Figure 4.4(b) as follows: `ctrl_x_5 = (N+1,0)`. Parameter β that corresponds to the iterator w is always initialized to 0 which indicates that the corresponding while-loop has not been executed.

Writing to the control variables is performed just after the writing to the corresponding data array. For example, control variable `ctrl_x_5` is written right after function $F5()$, see line 17 in Figure 4.4(b). This guarantees that when a function is executed, the current iteration is stored in a control variable. The value of control variable `ctrl_x_5` is propagated and assigned to the parameters α and β at line 22. These parameters are used to evaluate the source σ_x function at line 23 corresponding to

Solution 4.1 which determines the source for the data read by function *F7* at line 24. With the introduction of the control variables to the program shown in Figure 4.4(b), this program is input-output equivalent to the initial program in Figure 4.2(b).

```
1  #parameter EPS 0.005
2  w = 0
3  ctrl_x_5 = (N+1,0)
4  for i = 1 to N,
S1:  y_1[i] = F1()
6    in_2 = y_1[i]
S2:  x_2[i] = F2( in_2 )
W    while (in_w = σx(⟨W,(i,w)⟩) >= EPS),
9      w = w + 1
S3:    x_3[i,w] = F3()
11     for j = i+1 to N+1,
12       in_4 = σy(⟨S4,(i,w,j)⟩)
S4:      y_4[i,w,j] = F4( in_4 )
14       in_5_x = σx(⟨S5,(i,w,j)⟩)
15       in_5_y = y_4[i,w,j]
S5:      x_5[i,w,j] = F5( in_5_x, in_5_y )
17       ctrl_x_5 = (i,w)
18     endfor
19   in_6 = σx(⟨S6,(i,w)⟩)
S6:  y_6[i,w] = F6( in_6 )
21 endwhile
22 ctrl_x_5_[i] = ctrl_x_5

23 (α,β) = ctrl_x_5_[i]
24 in_7 = σx(⟨S7,(i,α,β)⟩)
S7: out = F7( in_7 )
26 endfor
```

Figure 4.6: Final dSAC.

4.5.1 Additional control variables

Unfortunately, introducing control variables to the dSAC code violates the property that "every variable is written *at most once*". For example, control variable `ctrl_x_5` that initializes parameter vector (α, β) at line 22 in Figure 4.4(b) is not in a single assignment form, i.e., `ctrl_x_5` may be written more than once at line 17. Therefore, the program in Figure 4.4(b) is not a dSAC anymore, and we cannot create a FIFO channel from control variable `ctrl_x_5`. In order to be able to create a process network, as discussed later in Section 4.6, and most importantly, to create the FIFO channels used for transferring control and data, the corresponding variables must be in a single assignment form.

In order to represent the program in Figure 4.4(b) as dSAC, we need to identify the relation between writing to and reading from the control variables. Thus, we need to perform dataflow analysis for the control variables, where the writings to them occur inside a while-loop. We achieve this in the following way. While keeping the same functionality, we introduce additional control variable `ctrl_x_5_` right *after* the while-loop, see line 22 in Figure 4.6. This program is input-output equivalent to the program in Figure 4.4(b). The new control variable is written at every iteration of *for*-loop `i` and takes the value either of control variable `ctrl_x_5` assigned on the last

iteration of the while-loop, or its initial value, if the while-loop is not executed. On this new control variable `ctrl_x_5_` we can perform the *static* exact array dataflow analysis (EADA) [4]. We can always do this, because the new control variable is not surrounded by the dynamic while-loop. The solution of EADA is used to modify the program in Figure 4.4(b) into the program in Figure 4.6 by inserting one-dimensional arrays `ctrl_x_5_[i]` at lines 22 and 23. The program in Figure 4.6 is in a dSAC form because the new control variable `ctrl_x_5_[]` used to initialize parameter vector (α, β) is in a single assignment form, thus allowing us to create a FIFO channel to communicate values of control variable `ctrl_x_5_[]`.

Finally, the program shown in Figure 4.6 is functionally equivalent to our running example shown in Figure 4.2(b). In the next step, we explain how to generate a process network from the program in Figure 4.6.

4.6 Step 4 (PPN generation)

Recall that a PPN consists of autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels. In this last step of our solution approach, we describe how the processes and FIFO channels are created from the corresponding final dSAC program derived in the previous step.

The procedure of PPN generation consists of 4 substeps. First, based on the final dSAC representation of a WLAP program derived in the previous step, the topology of the PPN is created. The topology is formed by instantiating processes and communication channels. Second, internal code structure of each process is derived from the dSAC specification. It is important to note, that in this substep, the created communication channels are not FIFOs but multi-dimensional arrays. Third, the multi-dimensional arrays that are used for data communication between function statements in the dSAC are replaced by FIFO channels. In other words, we replace the multi-dimensional array accesses in the code of each process with a read/write

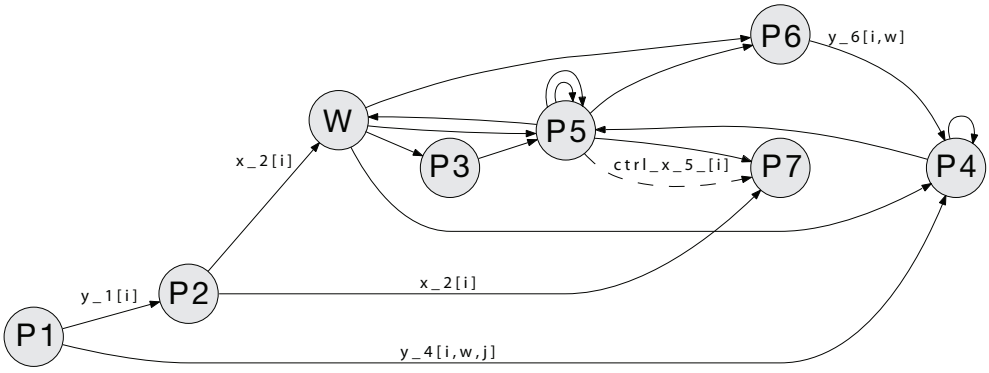


Figure 4.7: PPN representation of the program in Figure 4.6.

```
1  #parameter EPS 0.005
2  w = 0
3  for i = 1 to N,
4    while(1),
5      w = w + 1
6      if (w == 1),
7        in_w = x_2[i]
8      else
9        in_w = x_5[i,w-1,N+1]
10     end
11     C[i,w] = (in_w >= EPS)
12     if (!C[i,w]) <break>
13   endwhile
14 endfor
```

(a) Code of process W

```
1  w = 0
2  ctrl_x_5 = (N+1,0)
3  for i = 1 to N,
4   while(1),
5     w = w + 1
6     in_w = C[i,w]
7     if (!in_w) <break>
8     for j = i+1 to N+1,
9      if (j == i+1),
10      if (w == 1),
11       in_5_x = x_3[i,w]
12      else
13       in_5_x = x_5[i,w-1,N+1]
14      endif
15     else
16      in_5_x = x_5[i,w,j-1]
17     endif
18     in_5_y = y_4[i,w,j]
S5:    x_5[i,w,j] = F5( in_5_x, in_5_y )
20     ctrl_x_5 = (i,w)
21    endfor
22  endwhile
23  ctrl_x_5_[i] = ctrl_x_5
24 endfor
```

(b) Code of process P5

```
0  w = 0
2  for i = 1 to N,
3    (α,β) = ctrl_x_5_[i]
4    if (β>=1 && 1<= α <= i),
5       in_7 = x_5[α,β,N+1]
6    else
7       in_7 = x_2[i]
8    endif
S7:  out = F7( in_7 )
10 endfor
```

(c) Code of process P7

Figure 4.8: Internal source codes of processes W, $S5$ and $S7$.

primitives to implement synchronization through blocking read/write on FIFO communication channels. Fourth, the internal code structures of processes are modified to avoid the overflow of while-loop iterators which may lead to erroneous behavior of a PPN. Below, we explain the four substeps in more detail using the dSAC in Figure 4.6.

4.6.1 Substep 1: Topology creation of a PPN

The PPN that corresponds to the program in Figure 4.6 is depicted in Figure 4.7. This PPN consists of 8 processes and 18 channels. We explain how these processes and communication channels are created.

In our approach, one process is created for every function statement in the dSAC program, and one process is created for every while-loop's condition function. The latter process is needed to detect a while-loop's termination and notify the processes that execute functions enclosed in this while-loop. Therefore, the PPN in Figure 4.7 has 7 processes, $P1$–$P7$, that correspond to functions $F1$–$F7$ in Figure 4.6; and one process W which corresponds to the while-loop's condition function W at line 8 in Figure 4.6. The 18 communication channels correspond to data and control arrays in a single assignment form in the dSAC in Figure 4.6. Recall that data arrays in a single assignment are introduced after application of the FADA analysis on the WLAP program in Figure 4.2(b) as described in Step 1 of our solution approach. The control variables, i.e., array `ctrl_x_5_[i]` is introduced and transformed in a single

```
1  #parameter EPS 0.005
2  w = 0
3  for i = 1 to N,
4   while(1),
5    w = w + 1
6    if (w > 2) then w = 2
7    if (w == 1),
8     read(P2, 1, in_w)
9    else
10    read(P5, 2, in_w)
11   end
12   out_w = (in_w >= EPS)
13   write(P3, 3, out_w)
14   write(P4, 4, out_w)
15   write(P5, 5, out_w)
16   write(P6, 6, out_w)
17   if (!out_w) <break>
18  endwhile
19 endfor
```

(a) Code of process W

```
1  w = 0
2  ctrl_x_5 = (N+1,0)
3  for i = 1 to N,
4   while(1),
5    w = w + 1
6    if (w > 2) then w = 2
7    read(W, 1, in_w)
8    if (!in_w) <break>
9    for j = i+1 to N+1,
10    if (j == i+1),
11     if (w == 1),
12      read(P3, 2, in_5_x)
13     else
14      read(P5, 3, in_5_x)
15     endif
16    else
17     read(P5, 4, in_5_x)
18    endif
19    read(P4,5, in_5_y)
S5    out_5 = F5( in_5_x, in_5_y )
21    ctrl_x_5 = (i,w)
22    if (j == N+1),
23     write(P5, 6, out_5)
24    else
25     write(P5, 7, out_5)
26    endif
27   endfor
28  endwhile
29  out_5_c = ctrl_x_5
30  out_5_x = out_5
31  write(P7, 8, out_5_c)
32  write(P7, 9, out_5_x)
33 endfor
```

(b) Code of process P5

```
1  w = 0
2  for i = 1 to N,
3   read(P5, 1, in_c)
4   if (in_c.β>=1 && 1<= in_c.α <= i),
5    read(P5, 2, in_7)
6   else
7    read(P2, 3, in_7)
8   endif
S7: out = F7( in_7 )
10 endfor
```

(c) Code of process P7

Figure 4.9: Processes *W*, *P*5, and *P7* after linearization of multi-dimensional arrays.

assignment form in Step 3 of our solution approach. In the following substep, we describe how the internal code structure of each process is generated.

4.6.2 Substep 2: Code generation

Let us consider Figure 4.8, which illustrates the internal code structures of processes *W*, *P*5 and *P7* of the PPN in Figure 4.7. Process *W* is an example of a process detecting the termination of the while-loop at line 5 in Figure 4.2(b). Process *P*5 is an example of a process executing a function enclosed in the while-loop. Process *P7* is an example of a process that runs a function *outside* the while-loop and has a data dependency with a function inside the while-loop. Below, we will use them as examples to explain how the internal code structure of each process in the PPN is generated.

The internal code structure of each process is generated from the dSAC program derived in Step 3 of our solution approach. The code structure of each process is extracted from the code lines of the dSAC program. For example, all *non* highlighted

(non-bolded) code lines in Figure 4.8 are taken from dSAC in Figure 4.6 expanding all σ source functions as explained in Section 4.4 and illustrated in Figure 4.5. At this point, the PPN is not functionally equivalent to the dSAC program because for processes enclosed in a while-loop the termination problem is not solved yet.

To address this problem, process W is introduced which detects the termination of the while-loop. This process evaluates the while-loop's condition function and propagates the result to all processes that execute functions enclosed in this while-loop. This behavior is implemented in the highlighted (bolded) code at lines 4, 11 and 12 in Figure 4.8(a). Note, that lines 6–10 realize the interpretation of σ_x function defined in Solution 4.5 in Figure 4.3. A new array `C[i,w]` is added to propagate the value of the while-loop's condition function via FIFO to other processes. Correspondingly, we modify the code of process $P5$ in Figure 4.8(b) at lines 4, 6 and 7, where the information about while-loop termination is received and used. As process $P7$ executes function $F7$ which is outside the while-loop, no such modification is needed.

At this point, the processes of the PPN communicate data via multi-dimensional arrays. In the following substep, we explain how the multi-dimensional arrays are replaced with FIFO channels. This process is called *Linearization*.

4.6.3 Substep 3: Linearization

Processes W, $P5$ and $P7$ depicted in Figure 4.8 are connected with communication channels which are the multi-dimensional arrays inherited from the dSAC shown in Figure 4.6. However, the processes in our target PPN have to synchronize using a blocking read/write on an empty/full FIFO channel, i.e., an execution of a process is suspended if it tries to read from an empty FIFO channel, or tries to write to a full channel, respectively. Therefore, in order to synthesize a PPN, the multi-dimensional array accesses have to be replaced with corresponding *write* and *read* operations on FIFO channels. This is called "linearization".

To implement the Linearization, we adapted the approaches proposed in [29,54] and Chapter 5. In these works, the communication characteristics are identified when exchanging data between pair of statements. Based on this information, the multi-dimensional array accesses are replaced with one-dimensional array accesses. The result of the linearization applied on the arrays used in the internal source codes of the processes in Figure 4.8 is shown in Figure 4.9. In each process, the multi-dimensional arrays accesses are substituted by reading/writing primitives from/to FIFO channels. The communication read/write primitives access the FIFO channels through ports. That is, every process has a set of input ports and a set of output ports connected to FIFO channels. For example, process $P5$ in Figure 4.9(b) reads from process W and itself via ports 1, 3 and 4 at lines 7, 14 and 17. These input ports are connected with output port 5 of processes W, and output ports 6 and 7 of process $P5$, correspondingly. Internally, the read/write primitives realize the blocking synchronization between processes.

Additionally, we want to discuss how buffer sizes in FIFO channels of a PPN de-

rived from a WLAP program are determined. In our procedure we use the method of buffer sizes estimation presented in [3] and explained in Section 3.6 of this dissertation. Although this method accepts as an input a PPN derived from a *static* program, we explain how we adapt our procedure to use this method.

There are two types of channels in a PPN derived from a WLAP program: control and data channels. Control channels realize data dependencies between control variables. These dependencies are static and unique by construction. Therefore, we can safely use the method from [3] to determine buffer sizes in control channels. Data channels realize data dependencies between function statements of a program. In contrast to static programs, in WLAP programs data dependency relations are not static as some of the statements are enclosed in while loops. Therefore, the rate and the exact amount of data tokens that will be transferred over a particular data channel is unknown at compile-time, and we cannot directly use the method from [3] to determine buffer sizes.

However, with the following observation we are still able to determine buffer sizes. Consider two cases. First, if data dependency relation exists across a while-loop, i.e., a source statement is enclosed in the loop and the sink statement is outside, the while-loop acts as a barrier meaning that only the data from the last iteration of the while-loop has to be transferred to the sink. Therefore, in the code after a while-loop we can reconstruct a producer domain based on the data dependency relations with the data written on the last iteration of the while-loop. Next, we use the method from [3] to determine the buffer sizes of these data dependency relations. Second, if a data dependency relation exists between statements which are both enclosed in a while-loop, then based on Property 1 presented below in Section 4.6.4, and that w is not used in indexing we can use the method from [3] to determine the buffer sizes.

4.6.4 Substep 4: Implementation of a while-loop's iterator w

The PPN generated in the previous three substeps has a problem: potentially, iterator w may overflow the *finite* set of values determining the data type of the iterator. For example, if iterator w is specified by a 32-bit integer data type, the overflow may occur at line 5 in Figure 4.9(a) if the while-loop iterates more than 2^{32} times. As a consequence, it may lead to erroneous evaluation of the σ functions expanded in the previous code generation substep, and, finally, to erroneous behavior of a PPN. To address this problem, we show that it is sufficient to capture only 2 values of iterator w. To prove this, we use the following Property.

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 the same array. Both statements W and R are governed by a while-loop located at depth k.

Property 1 In the solution of the FADA algorithm applied on WR pair, the $k+1$-th dimension of mapping function $M(\vec{y})$ can be in one of the two forms: $\vec{y}[k+1]$ and $\vec{y}[k+1]-1$.

Proof: According to Property 1 in [37, 38], the solution defined by Equation 2.13 in Section 2.5 is exact, and iterator $\vec{y}[k+1]$ associated with the while-loop is present in sequencing predicate $(c3)$ only. Consider the expressions of $\mathbf{Q}^{p}_{WR}(\vec{y})$:

- If $k < p$, then the sequencing predicate includes $\vec{x}[1..k+1] = \vec{y}[1..k+1]$, and, thus, the lexicographical maximum of $\mathbf{Q}^{p}_{WR}(\vec{y})$ along $k+1$-th dimension is $\vec{y}[k+1]$.
- If $k = p$, then the sequencing predicate includes $\vec{x}[1..k] = \vec{y}[1..k] \wedge \vec{x}[k+1] < \vec{y}[k+1]$, and, thus, the lexicographical maximum of $\mathbf{Q}^{p}_{WR}(\vec{y})$ along $k+1$-th dimension is $\vec{y}[k+1] - 1$.

Initially, iterator w which is associated with a while-loop is initialized with value 0. This indicates that the while-loop has never been executed. From Property 1 and the fact, that only non-negative values of w determine source evaluations of statements enclosed in the while-loop, we conclude that it is needed to capture only 2 values of w: $w = 1$, meaning that the data dependency is at the same iteration of the while-loop; and $w \geq 2$, meaning that the dependency is at the previous iteration of the while-loop. The abovementioned reasoning allows us to modify the internal code structures of processes generated in the previous substep without altering their functionality. We introduce the code that captures only two values of iterator w. For example, see lines 6 in Figures 4.9(a) and 4.9(b).

4.7 Discussion and Summary

In this chapter, we presented an approach for automated translation of affine nested loops programs with while-loops (WLAPs) into input-output equivalent polyhedral process networks (PPNs). The approach presented in this chapter extended the work on an automated PPN derivation from a class of dynamic applications with more relaxed constrained than in Weakly Dynamic Programs (WDPs) presented in [21] and `Dynloop` programs presented in Chapter 3.

The work in [21] presented an approach for PPN derivation from Weakly Dynamic Programs (WDP). WDPs are more relaxed than the static class of applications because `if`-conditions might be dependent on some information that is unknown at compile-time and may change at run-time. In Chapter 3, we presented a first approach for automated translation of affine nested loops programs with dynamic loop bounds (`Dynloop`) into input-output equivalent PPNs. In this chapter, we further extended the class of applications to WLAP programs from which the PPN specification can be derived in an automated way.

The approach of PPN derivation from WLAP programs consists of the similar steps as the approach of PPN derivation from `Dynloop` programs: we apply Fuzzy Array Dataflow Analysis (FADA) on an initial program, transform the program into a dSAC specification and demonstrate how the parameters introduced by FADA are

set at run-time using control variables. Although the approaches of PPN generation from `Dynloop` and WLAP programs are similar, still, there are many important differences.

The first difference is that in order to analytically analyze the evaluation of a while-loop presented in a WLAP program, a new iterator w is introduced for every while-loop. An important consequence to this is that more parameters β are introduced to a dSAC specification and eventually it leads to more control FIFO channels present in a generated PPN.

The second difference is that in PPNs derived from WLAP programs we need to detect the termination of a while-loop and notify the processes in a PPN which execute functions enclosed in this while-loop. In order to handle this notification we introduce extra control channels that distribute the termination data in a specific way.

Another difference is that a PPN generated from a WLAP program has a problem: potentially, iterator w that corresponds to a while-loop may overflow the *finite* set of values determining the data type of the w iterator. In our approach presented in this chapter, we have shown that it is sufficient to capture only 2 values of iterator w and have proven this in Property 1 in Section 4.6.4. This property guarantees that we can implement iterator w in an efficient way that avoids the overflow. We have shown such implementation in Figure 4.9(a) and Figure 4.9(b).

All the above differences prove one more time that PPN derivation from WLAP programs is more difficult than PPN derivation from `Dynloop` programs. Information about the number of iterations of a while-loop in a WLAP program is unknown until the loop has been finished. Whereas the number of iterations of a for-loop with dynamic bounds in a `Dynloop` program is known just before the loop starts to execute. This absence of information in a while-loop requires much more advanced analysis compared to analysis of for-loops and, ultimately, produces a PPN with larger overhead.

The approach presented in this chapter includes only basic techniques that have to be applied in order to derive a PPN automatically from a WLAP program. Although, leveraging the FADA analysis this approach extracts the maximum parallelism available in an application, still, some optimization techniques have to be added to the approach that will help improving the quality of the generated PPNs in terms of optimal partitioning of the computation and communication workloads of a WLAP over processes and channels in the PPN. Our approach can be automated and implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential programs containing while-loops.

Chapter 5

Identifying Communication Models in Polyhedral Process Networks derived from Dynamic Programs

In Section 1.1, we have demonstrated that the Linearization is an important step of the parallelization approach depicted in Figure 1.2(b). As a result of the Dependence Analysis step, the initial program is translated into its functionally equivalent Polyhedral Reduced Dependence Graph (PRDG). The storage structure of the initial application is transformed such that each pair of statements communicates data over a dedicated multidimensional memory array as shown in Figure 1.2(c). However, the target model, Polyhedral Process Network (PPN), requires FIFO channel as communication medium. Therefore, the Linearization step converts such memory accesses into managed dataflow over FIFO queues.

Mapping array communication onto FIFO channels requires complex address generators, especially if the arrays have multiple dimensions. Therefore, the Linearization also solves the Communication Model Identification (CMI) problem, which investigates communication characteristics of each Producer/Consumer (P/C) pair.

In Section 1.1.2, we have demonstrated that there are four possible communication models that can describe the dataflow in a P/C pair. They are: *in-order (IO)*, *out-of-order (OO)*, *in-order with multiplicity (IOM)* and *out-of-order with multiplicity (OOM)*. Also, we have shown that the realization of different communication models requires different utilization resources (such as memory) and produces different runtime overhead. The difference in realization puts the communication models into a

hierarchy: from the most general *OOM* which can realize all communication models but requires most of the resources, to the specific, for example, *IO*, which can be realized as a FIFO in as straightforward way. Therefore, the identification procedure solves the optimization problem by finding the most optimal realization for each P/C pair.

In Section 2.6, in Definitions 2.6.1 and 2.6.3 we gave the overview of the Reordering and Multiplicity problems [14] which are used to identify communication models while translating *static* affine nested loop programs into functionally equivalent PPNs. In this chapter we present our novel compile-time procedure for communication model identification while translating the *dynamic* programs defined in Section 2.2 into functionally equivalent PPN. This procedure is used in the two parallelization approaches we have presented in Chapter 3 and Chapter 4.

In the following Section 5.1 we give the rationale of the communication model identification approach which will be presented in Section 5.2. Section 5.3 presents the conclusions.

5.1 Rationale

The overall challenge of Communication Model Identification while deriving a PPN from a dynamic program is how to deal with uncertainties introduced by the relaxations presented in Section 1.2. In Section 1.3 we have demonstrated that the approach used for CMI in static programs is inapplicable for dynamic programs.

In this dissertation, we use the PPN model of computation to specify both static and dynamic programs in parallel form. Therefore, the communication models in PPNs derived from dynamic programs are the same as in PPNs derived from static programs. These models are presented in Section 1.1.2.

Because the communication models in PPNs derived from static and dynamic programs are the same, we could have assumed that in order to identify the communication model in a P/C pair derived from a dynamic program we can use the RP and MP presented in Section 2.6. This problems are formulated according to Definitions 2.6.1 and 2.6.3 which give the formal definitions of ordering and multiplicity in a P/C pair. The key elements of these definitions are the mapping functions which are used to describe the dependency relations in a P/C pair. The definition of a mapping function is given in Definition 2.3.5.

However, there is a big difference in dependency relations between dynamic and static programs. In static programs, different instances of a program correspond to one and the same single dependency pattern which is known at compile-time. Therefore, only one unique set of mapping functions exist for all P/C pairs derived from a static program. We can observe this in Figure 1.3(b). In contrast to static programs, in dynamic programs, data dependency patterns correspond to different *instances* of a dynamic program, and are unknown at compile time. This has been demonstrated in Figure 1.6 explained in Section 1.3. Figure 1.6 depicts data dependency relations

between statements S1, S2 and S3 in three instances of the dynamic program in Figure 1.5(a) for $M = N = 4$. An instance of a dynamic program is an evaluation of the program with a particular input dataset. Figure 1.6 illustrates iteration domains of statements S1, S2 and S3, where the points on the coordinate systems designate the evaluations of statements and the arrows reflect the data dependencies between evaluations. The numbers at the points show the lexicographical order of statement evaluations.

Similar to static programs, from this figure it can be seen that for every instance of a dynamic program there is a unique set of mapping functions that determine the dependency relations between all statements for evaluation of the program with particular input data set. However, for different instances of a dynamic programs, there exist different sets of mapping functions that compensate for unknown, at compile-time, information inherent to dynamic programs.

To capture all unknown information at compile-time our novel approach is to define and use *parameterized* mapping functions that can be used to describe all possible dependency relations that exist in all instances of a dynamic program. By using these parameterized mapping functions, it is possible to analytically identify the most general communication model of a P/C pair in all possible instances of a dynamic program. Based on this information, the communication of a P/C pair can be realized with the most general communication model which implements all possible data dependency patterns occurring in all instances of a dynamic program.

In the following section we will demonstrate how these parameterized mapping functions are derived and how the parameters are determined at run-time. The derivation of parameterized mapping functions is our novel contribution.

5.2 Solution Approach

Because, the communication model identification procedure is a part of the Linearization step of the parallelization approaches presented in Chapters 3 and 4, we, first, briefly recall the main steps of these approaches. To illustrate these steps, we will use the running example shown in Figure 1.5(a).

The first steps of the parallelization approaches presented in Chapters 3 and 4 use the dependence analysis in order to extract dependency relations of the initial program. The Fuzzy Array Dependence Analysis (FADA) [37, 38] is used. The FADA is an enhanced version of Exact Array Dependence Analysis (EADA) [4] and it is used to analyze programs with dynamic behavior.

For example, consider the dynamic 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. Statement $S2$ is guarded by an `if`-condition whose values are determined at run-time. The FADA analysis of this program is described in Section 2.5. The result of the analysis is given below.

$$\sigma(\langle S3,(i_3,j_3)\rangle,(\alpha_i,\alpha_j)) = \left| \begin{array}{ll} \textbf{if} & i_3 \geq \alpha_i \wedge j_3 = \alpha_j \\ \textbf{then} & \langle S2,(\alpha_i,\alpha_j)\rangle \\ \textbf{else} & \langle S1,(j_3)\rangle \end{array} \right. \tag{5.1}$$

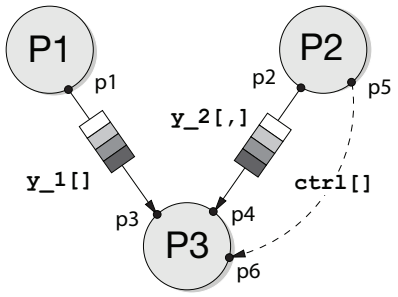
From Equation (5.1) it can bee seen that for any reading operation $\langle S3,(i_3,j_3)\rangle$ the source of the data can be from two different locations. The source is in $S1$ when for given j_3 none of the previous evaluations of the condition at line `C` in Figure 1.5(a) was `true`. Otherwise, the source is in $S2$. Parameters α_i and α_j are introduced by the FADA algorithm. Parameters are used to hide unknown, at compile-time, information and will be determined at run-time. Because the values of parameters are unknown at compile-time, the result of the dependence analysis (i.e., the source operation) shown in Equation (5.1) is unknown at compile-time either, and the result is approximated.

Based on the results of the FADA dependence analysis, the initial sequential program is translated into a *dynamic* Single Assignment Code (dSAC) representation of the initial dynamic program. The dSAC was proposed in [21] as an extension of the Single Assignment Code [4]. A dSAC program is input-output equivalent to the corresponding initial dynamic program and it has the property that every data variable or an array element is written *at most once*. This implies that some variables may not be written at all.

```
1  for j = 1 to 4,
2    ctrl[j] = (5,5)
3  end
4  for k = 1 to 4,
S1:  y_1[k] = F1()
6  end
7  for i = 1 to 4,
8    for j = i to 4,
C:     if y_1[j]<=2,
S2:     y_2[i,j] = F2()
11      ctrl[j] = (i,j)
12     end
13     c1 = ctrl[j].i
14     c2 = ctrl[j].j

15     if i >= c1 && j == c2,
16       in_0 = y_2[c1, c2]
17     else
18       in_0 = y_1[j]
19     end
S3:    [] = F3(in_0)
21   end
22 end
```

(a) The dSAC form



(b) PPN specification

Figure 5.1: An example of a dSAC program and corresponding PPN specification derived from the program shown in Figure 1.5(a).

For example, the dSAC form of the program in Figure 1.5(a) derived from Equation 5.1 is depicted in Figure 5.1(a) where the parameters N and M are set to 4. It is in dSAC form because if we consider line 10, we do not know at compile-time at which iteration the elements of array `y_2[]` will be written. The only thing known is that they will be written at most once. The other array `y_1[]` has the same property as an array in SAC form: every element is written exactly once.

Another property of the dSAC form is the presence of parameters that originate from the FADA analysis. For example, in Figure 5.1(a) the program variables are `c1` and `c2` which correspond to parameters α_i and α_j in Equation 5.1. In order to make dSAC to be functionally equivalent to the initial dynamic program, the values of these parameters have to be changed at run-time.

Parameters are changed with the help of control variables that store the correct value of the parameters for every iteration. For example, dynamic change of the values of `c1` and `c2` is accomplished by lines 13 and 14. The control array `ctrl[]` at line 11 stores the iterations for which the data-dependent condition at line *C* is true. The control variables must be initialized with values that are greater than the maximum value of the corresponding parameters. Therefore, the values of control array `ctrl[]` are initialized with value `(5,5)` at lines 1–3 in Figure 5.1(a). Writing to the control variables is performed just after the functional statement $F2()$, see line 11 in Figure 5.1(a). This guarantees that when the function is executed, the current iteration is stored in the control variables. The values of the control variables are propagated and assigned to the program variables `c1` and `c2` at lines 13 and 14. These variables are used to evaluate the conditions at lines 15 and 16 which determine the source of the data for function $F3()$.

From the dSAC we can build the topology of the PPN depicted in Figure 5.1(b). Every functional statement becomes a process, and every variable or array becomes a channel. For example, lines 4-6 form process P1; lines 1-3 and 7-12 form process P2; and, finally, lines 7-8, 13-21 form process P3. Processes P1 and P3 are connected with a FIFO channel via ports `p1` and `p3`. Processes P2 and P3 are connected with two FIFO channels: the first one (ports `p2` and `p4`) originated from array `y_2[,]` for transferring data, and the second channel which originates from control variable `ctrl[]`, to communicate the outcome of the condition at line C.

5.2.1 Parameterized mapping functions

From the dSAC form and Equation 5.1 we derive parameterized mapping functions which are functions of the Consumer iteration point and vectors of parameters: $f(\vec{y}, \vec{\alpha})$. Vector of parameters $\vec{\alpha}$ is used to uniformly specify a set of unique mapping functions which exist for every instance of a dynamic program, thus capturing the unknown information at compile-time. Values of vector $\vec{\alpha}$ will be determined at run-time during the execution of a PPN.

We define a parameterized mapping function as follows:

Definition 5.2.1 (parameterized mapping function)
A parameterized mapping function is an affine mapping $f_{pq} : I_p \rightarrow O_q : O_q = f(I_p, \vec{\alpha})$, where $I_p \in IPD_p(\vec{\alpha})$ and $O_q \in OPD_q$.

For example, for the P1/P3 pair (ports `p1-p3`) in the PPN shown in Figure 5.1(b), the parameterized mapping function and its domain are:

$$f_{p3p1} : \mathbb{Z}^4 \rightarrow \mathbb{Z} : k = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} (i_3, j_3, \alpha_i, \alpha_j)^{\mathbf{t}}, \tag{5.2}$$
$$\mathbf{D}(f_{p3p1}) = \{(i_3, j_3, \alpha_i, \alpha_j) \in \mathbb{Z}^4 \mid 1 \leq i_3 < \alpha_i \leq 4 \vee i_3 \leq j_3 \neq \alpha_j \leq 4\}.$$

For the P2/P3 pair (ports `p2-p4`), the parameterized mapping function and its domains are:

$$f_{p4p2} : \mathbb{Z}^4 \rightarrow \mathbb{Z}^2 : (i_2, j_2)^{\mathbf{t}} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (i_3, j_3, \alpha_i, \alpha_j)^{\mathbf{t}}, \tag{5.3}$$
$$\mathbf{D}(f_{p4p2}) = \{(i_3, j_3, \alpha_i, \alpha_j) \in \mathbb{Z}^4 \mid 1 \leq i_3 \geq \alpha_i \leq 4 \wedge i_3 \leq j_3 = \alpha_j \leq 4\}.$$

Finally, we use the parameterized mapping functions to formulate the *Reordering Problem* (RP) and the *Multiplicity Problem* (MP) used to identify the communication models in a PPN derived from a dynamic program. This problems are shown in Figure 5.2 and correspond to Definitions 2.6.1 and 2.6.3, respectively.

The meanings of all constraints in Figure 5.2 are the same as in Figure 2.3. A major difference is that *parameterized* mapping functions are used. The novelty of our communication model identification procedure is to model unknown information at compile-time by the parameterized mapping functions.

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

(a) Reordering Problem (RP)

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

(b) Multiplicity Problem (MP)

Figure 5.2: Reordering and Multiplicity Problems for used in communication model identification in PPNs derived from dynamic programs.

The definition of the LmP set used in Figure 5.2(a) is given in Figure 5.3(a). It is a parametric integer linear problem (PILP) similar to the problem given in Definition 2.6.2. The differences between the two formulations of LmP problems are that in the problem shown in Figure 5.3(a) the mapping function is parameterized and this problem finds lexicographically minimal Consumer's iteration points *and* parameter vector $\vec{\alpha}$. For example, consider the P2/P3 pair (and its corresponding ports `p2` and `p4`) formed by statements S2 and S3 from the dSAC shown in Figure 5.1(a).

The LmP problem for the P2/P3 pair is illustrated in Figure 5.3(b). The solution of this problem is $\text{LmP}(\mathbf{D}(f_{p4p2})) = \{(i_3, j_3, \alpha_i, \alpha_j) \in \mathbb{Z}^4 \mid i_3 = i_2 \wedge j_3 = j_2 \wedge \alpha_i = i_2 \wedge \alpha_j = j_2 \wedge 1 \leq i_2 \leq j_2 \leq 4\}$.

objective :

$$(\vec{y}_m, \vec{\alpha}_m) = lexmin\ \{f^{-1}(\vec{x})\},$$

subject to :

$$\begin{cases} \vec{y} \in \mathbf{D}(f), \\ \vec{x} = f(\vec{y}, \vec{\alpha}). \end{cases}$$

(a) Formulation of the LmP problem

$$\begin{cases} i_3 \geq \alpha_i, j_3 = \alpha_j, \\ i_2 = \alpha_i, j_2 = \alpha_j, \\ 1 \leq i_3 \leq j_3 \leq 4, \\ 1 \leq i_2 \leq j_2 \leq 4. \end{cases}$$

(b) An example of LmP problem

Figure 5.3: Formulation of the problem with an example used to find Lexicographically minimal Preimage (LmP) of the Consumer iteration points while deriving a PPN from a dynamic program.

Examples of applying RP and MP problems to our running example depicted in Figure 1.5(a) are shown in Figure 5.4. The RP and MP problems are formulated for the P2/P3 pair formed by statements S2 and S3 from the Figure 5.1(a), respectively. Clearly, the RP problem shown in Figure 5.4(a) does not have an integer solution, because constraints $(c3)$ and $(c4)$ contradict each other. Therefore, the communication model of P2/P3 pair is **in-order**. The MP problem illustrated in Figure 5.4(b) has an integer solution: for some $i_3^1 \neq i_3^2$, there can be $\alpha_i^1 = \alpha_i^2$ which satisfy $(c4)$, and, thus, the communication model of the channel has a **multiplicity**. Therefore, the communication model of P2/P3 pair is *IOM*.

$$\begin{cases} i_3^1 = \alpha_i^1, j_3^1 = \alpha_j^1, \\ i_3^2 = \alpha_i^2, j_3^2 = \alpha_j^2, \\ (i_3^1, j_3^1) \ll (i_3^2, j_3^2), \\ (\alpha_i^1, \alpha_j^1) \gg (\alpha_i^2, \alpha_j^2). \end{cases}$$

(a) RP for P2/P3 pair

$$\begin{cases} i_3^1 \geq \alpha_i^1, j_3^1 = \alpha_j^1, & (c1) \\ i_3^2 \geq \alpha_i^2, j_3^2 = \alpha_j^2, & (c2) \\ (i_3^1, j_3^1) \neq (i_3^2, j_3^2), & (c3) \\ (\alpha_i^1, \alpha_j^1) = (\alpha_i^2, \alpha_j^2). & (c4) \end{cases}$$

(b) MP for P2/P3 pair

Figure 5.4: Examples of RP and MP problems for P2/P3 pair.

5.3 Discussion and Summary

In this chapter, we have presented a novel procedure for communication models identification while deriving PPNs from the dynamic programs defined in Section 2.2. The procedure, needed to convert multidimensional arrays used for data communication after the Data Dependence analysis step, distinguishes between four patterns of communicating data and identifies the most general one which can realize all dependency patterns in different instances of a dynamic program. This identification procedure solves the optimization problem by finding the most optimal realization for each P/C pair.

The novel idea that we have used in our communication model identification procedure is the parameterized mapping functions. We derive them from a dSAC specification of an initial dynamic application using the Fuzzy Array Dataflow Analysis. Parameterized mapping functions are used to describe all possible dependency relations that exist in all instances of a dynamic program for each P/C pair. Parameters in these functions are used to uniformly specify a set of unique mapping functions which exist for every instance of dynamic program, thus, capturing unknown information at compile-time. Using parameterized mapping functions, we reformulated the problems presented in Section 2.6: the *Reordering Problem* (RP) and the *Multiplicity Problem* (MP) which are used to identify communication models in a derived PPN.

Chapter 6

Experimental Studies

In this chapter, we examine the approach presented in Chapter 3 by deriving a PPN parallel specification from a real-life application called Low Speed Obstacle Detection (LSOD). This application contains relaxation II described in Section 1.2. We present some of the results we have obtained by implementing and executing the derived parallel PPN specification of the LSOD application on a shared memory multi-processor system. The main objective of this experiment is to show the practical applicability of our approach on a real-life application and to show the benefits of our parallelization approach. For the implementation, we derive the PPN specification of the LSOD application following the approach presented in Chapter 3. Then, we use the ESPAM [55] tool and the HDPC [56] back-end library to generate C++ code for the processes and the FIFO channels.

In this chapter, in Section 6.1 we evaluate our parallelization approach presented in Chapter 3, and in Section 6.2 we present the conclusions.

6.1 Low Speed Obstacle Detection

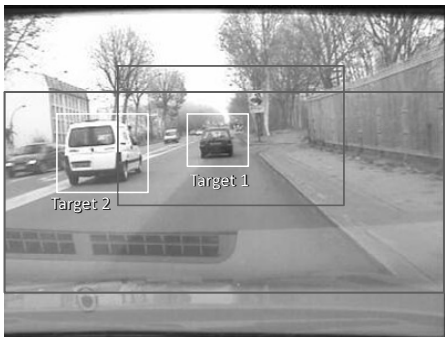
The LSOD application shown in Figure 6.1(a) is intended to detect and to track objects in front of a car in traffic. The output of the system presents spatial positions for targets – cars, pedestrians, etc. Applying several general image processing algorithms helps to find new targets, and to track existing targets. The algorithms implement shadow detection, symmetry detection, lights detection, motion segmentation, and vertical edge detection. The output of each algorithm is collected by a particle filter component [24] for analysis.

The first step in the LSOD application is to obtain two images from a given camera picture. They are named high and low resolution images and are depicted by the

two dark rectangles in Figure 6.1(b). Applying different image processing algorithms on these images, hypotheses whether cars exist are computed. Possible targets are defined as coordinates and dimensions of rectangles belonging either to the high or low resolution image. In Figure 6.1(b), two possible targets are presented by the white rectangles, surrounding the cars. Then, for every identified target, the image gradient in vertical direction of the area of the target is computed. The result is finally analyzed in order to support or decline a target.

```
0 vsum[] = 0
1 for k = 1 to Targets,
2   [Height,Width,X,Y] = getLSODTarget(k)
3   for j = 0 to Height+1,
4     for i = 0 to Width+1,
5       img[j,i] = readTarget(X,Y)
6     endfor
7   endfor
8   for j = 1 to Height,
9     for i = 1 to Width,
10      img_out[j,i] = edgeDetection(
                       img[j-1,i-1],img[j-1,i+1],
                       img[j  ,i-1],img[j  ,i+1],
                       img[j+1,i-1],img[j+1,i+1])
11      img_out[j,i] = absVal( img_out[j,i] )
12    endfor
13  endfor
14  for j = 1 to Height,
15    for i = 1 to Width,
16      vsum[i] = vertSum( vsum[i], img_out[j,i] )
17    endfor
18  endfor
19 endfor
```

(a) Pseudo code of the edge detection part of the motivating example. Target size is specified by variables `Height` and `Width`.



(b) LSOD applied on real data. The vehicles in front of the camera are detected and tracked. The dark rectangles depict the area of the image that is processed.

Figure 6.1: Pseudo code of the edge detection part of the LSOD application and its application on real data.

The edge detection part of the LSOD application, shown in Figure 6.1(a), is an example of a program which is not a static affine nested loop program. This program is affine nested loop program but it has dynamic control as function `getLSODTarget()` at line 2 initializes variables `Height` and `Width` used as loop bounds. These variables define the size of a target, i.e., the amount of data to be processed, and change values for every target at run-time. Since targets are moving in front of a camera (which is also moving), the identified positions stored in variables `(X,Y)` and dimensions `(Height,Width)` will differ for different targets in the frame and for one and the same target in different frames. That is why, the values of variables `Height` and `Width` (as well as the number of targets) are unknown at compile-time, and therefore, the *pn* compiler [3] cannot handle the program shown in Figure 6.1(a).

Parallelization

The result of Steps 1 to 3 of the solution approach presented in Chapter 3 and applied on the LSOD program in Figure 6.1(a) is illustrated in Figure 6.2. It is the final dSAC specification. We use the final dSAC to derive the PPN topology, shown in Figure 6.3, as well as to derive the internal code structure of all processes. Examples of the internal code structures of processes `readTarget` and `edgeDetection` are depicted in Figure 6.4. Below, we emphasize on some of the important moments of the PPN derivation, we describe the PPN topology, show how code for processes is generated, and comment on the overhead introduced in the generated PPN.

```
1  for k = 1 to Targets,
2    [Height,Width] = getLSODTarget()
3    X_j[k] = Height
4    X_i[k] = Width
5    for j = 0 to maxHeight + 1,
6      for i = 0 to maxWidth + 1,
7        if (j <= X_j[k] + 1 && i <= X_i[k] + 1)
8          img_1[k,j,i] = readTarget(X,Y)
9          lcl_1[j,i] = (j,i)
10       endif
11       ctrl_1[k,j,i] = lcl_1[j,i]
12     endfor
13   endfor
14   for j = 1 to maxHeight,
15     for i = 1 to maxWidth,
16       (c11,c12) = ctrl_1[k,j-1,i-1]
17       if (c11 == j-1 && c12 == i-1)
18         in_0 = img_1[k,j,i]
19       endif
20       (c21,c22) = ctrl_1[k,j-1,i+1]
21       if (c21 == j-1 && c22 == i+1)
22         in_1 = img_1[k,j,i]
23       endif
24       (c31,c32) = ctrl_1[k,j,i-1]
25       if (c31 == j && c32 == i-1)
26         in_1 = img_1[k,j,i]
27       endif
28       (c41,c42) = ctrl_1[k,j,i+1]
29       if (c41 == j && c42 == i+1)
30         in_1 = img_1[k,j,i]
31       endif
32       (c51,c52) = ctrl_1[k,j+1,i-1]
33       if (c51 == j+1 && c52 == i-1)
34         in_1 = img_1[k,j,i]
35       endif
36       (c61,c62) = ctrl_1[k,j+1,i+1]
37       if (c61 == j+1 && c62 == i+1)
38         in_5 = img_1[k,j,i]
39       endif
40       if (j <= X_j[k] && i <= X_i[k])
41         img_out_1[k,j,i] = edgeDetection(
                                 in_0, in_1,
                                 in_2, in_3,
                                 in_4, in_5)
42         img_out_2[k,j,i] = absVal( img_out_1[k,j,i] )
43         lcl_2[j,i] = (j,i)
44       endif
45       ctrl_2[k,j,i] = lcl_2[j,i]
46     endfor
47   endfor
48   for j = 1 to maxHeight,
49     for i = 1 to maxWidth,
50       (c71,c72) = ctrl_2[k,j,i]
51       if (j == c71 && i == c72)
52         in_0 = img_out_2[k,j,i]
53       endif
54       if (j <= X_j[k] && i <= X_i[k])
55         if( j >= 1 )
56           in_1 = vsum[i]
57         else
58           in_1 = 0
59         endif
60         vsum[i] = vertSum( in_1, in_0 )
61       endif
62     endfor
63   endfor
64 endfor
```

Figure 6.2: Final dSAC.

According to Step 1 (see Section 3.2) of our solution approach we substitute the dynamic upper bound functions with constants equal to the maximum values these functions can have. The initial LSOD program in Figure 6.1(a) has three loop nests with dynamic upper bound functions: `Height+1`, `Width+1`, `Height` and `Width` at lines 3–4, 8–9 and 14–15, respectively. These functions take their maximum when variables `Height` and `Width` are maximum, i.e., equal to some constants `maxHeight` and `maxWidth`, respectively. In the LSOD program, the maximum values of `Height` and `Width` are the maximum dimensions that a target may have. Therefore, we substitute

the dynamic upper bound functions with constants equal to the maximum of these functions as depicted in Figure 6.2 at lines 5–6, 14–15 and 48–49.

The result of applying the FADA analysis which constitutes Step 2 (see Section 3.3) of our solution approach is illustrated at lines 17–19, 21–23, 25–27, 29–31, 33-35, 37–39 and 51-53 in Figure 6.2. In total, 6 two-dimensional vectors of parameters $(c11, c12), \ldots, (c61, c62)$ were introduced by the FADA algorithm analyzing the data-dependencies between functions `readTarget()` and `edgeDetection()` shown in Figure 6.1(a). Also, one two-dimensional vector of parameters $(c71, c72)$ was introduced after analyzing the data-dependencies between functions `absVal()` and `vertSum()`.

At Step 3 (see Section 3.4) of our solution approach, we introduce local and global control arrays in order to initialize and propagate the values of the parameters at run-time. For the pair of functions `readTarget()` and `edgeDetection()`, 6 vectors of parameters were introduced by FADA. All these parameter vectors correspond to the single iteration vector (j_2, i_2) of the source function `readTarget()`. Therefore, at lines 9 and 11 only one local and one global control arrays are generated for this pair of functions. Similarly, at lines 43 and 45 one local and one global control arrays are added for the pair of functions `absVal()` and `vertSum()`.

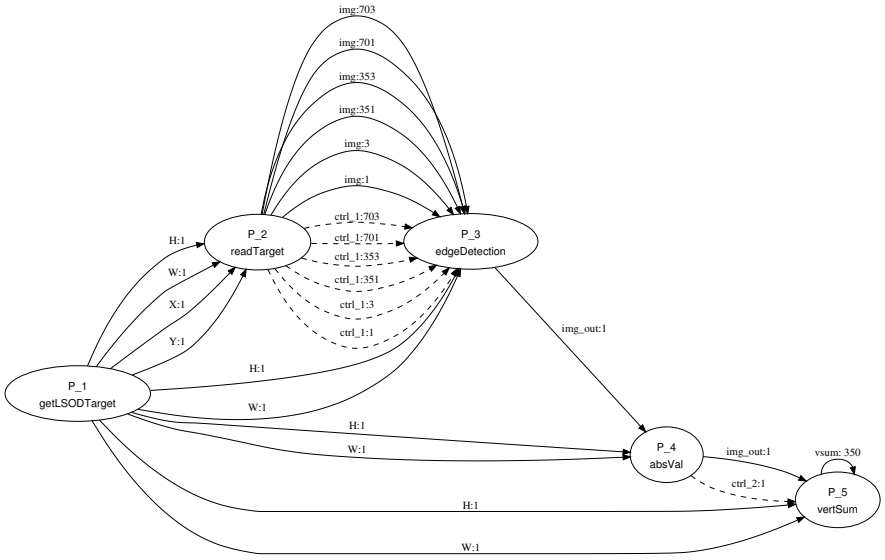


Figure 6.3: The PPN derived from the LSOD program.

By applying Step 4 (see Section 3.5) of our approach to the final dSAC of the LSOD application depicted in Figure 6.2, we derive the topology of the PPN depicted in Figure 6.3. The topology consists of 5 processes, 19 *data channels* shown as solid lines that are used to exchange data between processes and 7 *control channels* shown as dashed lines used to propagate values of global control arrays. The edges of the PPN in Figure 6.3 are annotated with the buffer sizes calculated for the LSOD program considering maximum dimensions of the targets to be 350x300 pixels. This means

that we set maxWidth = 350 and maxHeight = 300.

Finally, as an example of the internal structure of the PPN processes, in Figure 6.4, we present the pseudo code for readTarget and edgeDetection processes derived after the linearization step. These processes exhibit the most intensive data communication in the PPN. The internal code structures of these processes are generated as it has been explained in Step 4 (see Section 3.5) of our solution approach. Note, that the input/output ports used to access FIFO channels (see the read/write primitives in Figure 6.4) are automatically mapped to physical addresses generated by the Espam tool (in a separate header file).

```
1  for k = 1 to Targets,
2    for j = 0 to maxHeight + 1,
3      for i = 0 to maxWidth + 1,
4        if (j == 0 && i == 0)
5          read(0, H)
6          read(1, W)
7          read(2, X)
8          read(3, Y)
9        endif
19       if (j <= H + 1 && i <= W + 1)
11         img_1 = readTarget(X,Y)
12         lcl_1 = (j,i)
13         if( j <= H-1 && i <= W-1 )
14           write( 1, img_1 )
15         if( j <= H-1 && i >= 2 )
16           write( 3, img_1 )
           ...
17       endif
18       ctrl_1 = lcl_1
         if( j <= H-1 && i <= W-1 )
           write( 0, ctrl_1 )
         if( j <= H-1  && i >= 2 )
           write( 1, ctrl_1 )
         ...
23     endfor
24   endfor
25 endfor
```

(a) Process readTarget

```
1  for k = 1 to Targets,
2    for j = 1 to maxHeight,
3      for i = 1 to maxWidth,
4        if (j == 0 && i == 0)
5          read(0, H)
6          read(1, W)
7        endif
8        if (j <= H && i <= W)
9          read(0, ctrl_1)
10         if (ctrl_1.j == j-1 &&
               ctrl_1.i == i-1)
11           read(1, in_0)
12         read(2, ctrl_1)
13         if (ctrl_1.j == j-1 &&
               ctrl_1.i == i+1)
14           read(3, in_1)
           ...
15         img_out = edgeDetection(
                     in_0, in_1,
                     in_2, in_3,
                     in_4, in_5)
           write(0, img_out)
         endif
19     endfor
20   endfor
21 endfor
```

(b) Process edgeDetection

Figure 6.4: Internal code structures of processes readTarget and edgeDetection of the PPN derived from the LSOD application.

We evaluate our approach by executing the parallel LSOD application specification on an Intel® Xeon® quad-core machine running a Linux operating system. We used the ESPAM [55] tool and the HDPC [56] library to map the processes of the generated PPN onto cores and to generate C++ code for these cores. We used the GCC compiler to generate the final binary code. The HDPC library employs the boost-thread framework that enables the use of multi-threaded implementations. That is, the derived PPN has been translated to a multi-threaded program realizing the LSOD application, in which every process of the PPN is a separate thread.

In this experiment, we implemented and executed the parallel PPN specification of the LSOD application considering 5 different mapping configurations. The obtained results are shown in Figure 6.5. The horizontal axis depicts the number of cores used

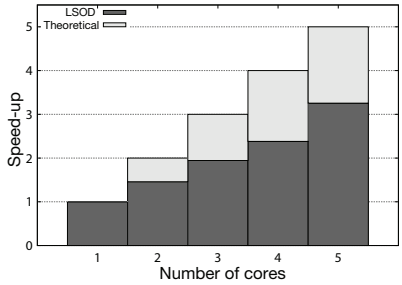


Figure 6.5: Evaluation of LSOD PPN on several CPUs.

in each configuration, i.e., we mapped the 5 processes of the PPN onto 1, 2, 3, 4, and 5 cores, respectively. Note that because the `Intel® Xeon®` processors support hyper-threading, the operating systems 'sees' 8 different cores. Therefore, we could map 5 processes on 5 different cores exploiting maximum concurrency. Obviously, when using less than 5 cores, some processes have to share the same core. In this case, we let the operating system to schedule the execution of these processes (i.e., threads) on a particular core. We experimented with grouping different processes together, i.e., mapping several processes on a single core. Figure 6.5 presents the best results that we have obtained. In addition, every configuration has been executed multiple times and the bars show the obtained average speed-up. The first configuration (see the leftmost bar in Figure 6.5) represents our reference configuration, in which, we mapped all 5 processes of the PPN onto a single core. We consider the speed-up of this configuration to be 1. Also, we have normalized the performance of the other configurations with respect to the performance of this reference configuration. Looking at the performance of the other 4 configurations, we see that by increasing the number of cores, the speed-up increases below the theoretical maximum shown as gray bars in Figure 6.5. We found that because of the data dependencies in the LSOD application and the imbalanced workload of the functions executed by different processes, the theoretical maximum cannot be achieved. In addition, in all configurations except the one using 5 cores (see the rightmost bar in Figure 6.5), there is an overhead introduced by the operating system for scheduling different threads on one core. As a result, the rightmost configuration exhibits a slightly larger speed-up increase compared to the other configurations. Finally, there is the execution time overhead caused by the extra 'dummy' iterations, which we discussed in Section 3.7. Below, we present some numbers with regards to this execution time overhead, as well as, the memory overhead in the LSOD process network.

Execution time overhead

From the execution statistics obtained by profiling of the LSOD application and its PPN, we computed the two ratios in Equation 3.4 presented in Section 3.7. Table 6.1 shows the ratios for each process and the whole PPN. Note that for computing $(max\_f - x)/x$, we need to consider that the targets are 2-dimensional. Therefore,

we used the term:

$$\frac{maxWidth \cdot maxHeight - x \cdot y}{x \cdot y} = \frac{350 \cdot 300 - x \cdot y}{x \cdot y}.$$

The terms x and y represent the average target size, which we computed from the targets used when executing the program. Based on the computed values in the last column of Table 6.1 and applying Equation 3.4 in Section 3.7, the overhead due to the execution of 'dummy' iterations of the LSOD PPN is 33.93%.

Process	P_1	P_2	P_3	P_4	P_5	PPN
$W/(W_x + W)$	0.53	0.38	0.26	0.47	0.3	0.39
$(max\_f - x)/x$	0	1.09	1.09	1.09	1.09	0.87

Table 6.1: Statistics LSOD PPN.

Memory overhead

In order to evaluate the memory overhead, we measured the memory requirements for the sequential LSOD program and compared this with the memory requirements for executing the corresponding PPN. The sequential program requires in total 13650 Bytes of memory, which includes both the code and the data. In order to make a fair comparison, it is important to note that this number (13650 Bytes) does not include the data array used to buffer the largest possible target, i.e., variable `img[TH][TW]` which requires 350x300 = 105000 Bytes. Although, we use such a variable in the sequential program, the program can be written more efficiently in a way that we do not need to buffer the whole (largest possible) target. The left part of Table 6.2 shows the memory requirements for every process in the generated process network. In addition, we need to consider also the memory used to implement the FIFO channels. In total, the PPN requires 17018 Bytes for implementing the processes and 18384 Bytes for the FIFO channels, see the right part of Figure 6.2. Then, if we compare these numbers with the number of the sequential program, we see that the memory overhead is 2.6x, which is reasonable provided that this is the memory requirement for the implementation of 5 processes and 26 FIFO channels.

Process	P_1	P_2	P_3	P_4	P_5
Code (bytes)	1626	2302	2510	1742	1978
Data (bytes)	1420	1360	1360	1360	1360
Total (bytes)	3046	3662	3870	3102	3338

	PPN	Sequential
Memory (bytes)	17018	13650
FIFOs (bytes)	18384	–
Overhead	2.6x	–

Table 6.2: Memory overhead of the LSOD PPN.

Overall, the average efficiency of the 4 parallel implementations of the LSOD process network is around 70%. The efficiency (Eff) is defined as:

$$Eff = \frac{SP}{C},$$

where SP is the speed-up and C represent the number of cores used to achieve this speed-up.

6.2 Discussion and Summary

In this chapter, we evaluated our parallelization approach presented in Chapter 3 on a real-life application called Low Speed Obstacle Detection (LSOD). This application contains for-loops with dynamic bounds and is an example of an application with relaxation II presented in Section 1.2. By evaluation of this application, we demonstrated the practical applicability of our parallelization approach to a real-life dynamic application.

By evaluating our approach we found that because of the data dependencies in the LSOD application and the imbalanced workload of the functions executed by different processes, the theoretical maximum cannot be achieved. In addition, there are two types of overhead in the generated PPN, i.e., memory and execution time overhead. The execution overhead is caused by the execution of some 'dummy' iterations not present in the initial sequential program. The memory overhead is due to the introduced control arrays, as well as, the created dataflow and control FIFO channels. It highly depends on the characteristics of the application being parallelized. In Section 3.7, we presented analytical analysis of the execution overhead. For the LSOD application, the overhead due to the execution of 'dummy' iterations of the LSOD PPN is 33.93%. This overhead is highly dependent on the maximum dimensions of the targets in the image.

In order to evaluate the memory overhead, we measured the memory requirements for the sequential LSOD program and compared this with the memory requirements for executing the corresponding PPN. The total memory overhead is 2.6x which is reasonable, because this is the memory requirement to implement 5 separate processes and 26 FIFO channels which is unavoidable if we want to parallelize the LSOD to the maximum task-level parallelism available in the initial LSOD specification. Overall, the average efficiency of the 4 parallel implementations of the LSOD process network is around 70% which is very reasonable for parallel implementation.

From the LSOD evaluation, the obtained results indicate that the approach we presented in Chapter 3 facilitates efficient parallel implementations of sequential nested loop programs with dynamic loop bounds. That is, our approach reveals the possible parallelism available in such applications, which allows for the utilization of multiple cores in an efficient way.

Chapter 7

Summary and Conclusions

In this dissertation, we addressed the problem of automated derivation of Polyhedral Process Network (PPN) specifications from *dynamc* sequential programs. Our work is essential for the development of parallelization compilers exploiting task-level parallelism inherent to many dynamic applications. As an example, the work presented in this dissertation inspired the development of further extensions in the *pn* [3, 48] compiler, where at the moment, most of the research done in this dissertation has been implemented. The derivation of a parallel specification described by our approach is an essential for the systematic and automated design of the emerging embedded systems-on-chip platforms. In designing the platforms the parallel specification allows for systematic and efficient exploration and mapping of the application onto the platform. As an example, the work presented in this dissertation is used in a methodology, implemented in a system-level design tool-flow called DAEDALUS [57, 58], for automated design, programming, and implementation of MPSoCs starting at a high level of abstraction. The methodology is built on the concept of Platform-Based Design (PBD) [59] being a promising new approach to master the ever growing complexity of today's embedded systems.

Many system-level design flows and application modeling and exploration approaches reported in the literature use the Kahn Process Network (KPN) [7] model of computation for a parallel application specification. In this dissertation, we target Polyhedral Process Networks (PPNs) [6] which is a special case of the KPN model. The PPN allows to specify an application, manipulate and optimize its representation in terms of polyhedra. This model is well suited for data-flow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. In this dissertation, we target such applications as being natural for extracting task-level parallelism.

The work presented in this dissertation is directly related to previous works on systematic and automated derivation of process networks from affine nested loops pro-

grams initiated by Rijpkema et al. [15, 30]. Further, Turjan et al. [14] proposed an automated derivation of process networks from a class of application called *static* affine nested loop programs (SANLPs). In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Also, they put a restriction on the input program to be *static* in order to enable the automatic analysis and conversion of the input program to a PPN. Although, many scientific, matrix computation, and signal processing applications can be specified as SANLPs, the *static* restriction limits the applicability of their approach, i.e., their approach cannot handle applications that have adaptive and dynamic behavior, such as multimedia applications (MPEG coders/decoders, Smart Cameras, Software Radio), adaptive filters, iterative algorithms, etc. Therefore, in this dissertation, we addressed the important question: whether some of the static restrictions of the SANLPs can be relaxed while keeping the ability to perform compile-time analysis and to derive PPNs in an automated way. Achieving this would significantly extend the range of applications that can be parallelized in an automated way.

By studying different dynamic applications we distinguished three relaxations to SANLP programs that would allow one to specify dynamic applications as sequential programs. These relaxations are:

I. dynamic `if`-conditions are allowed in a program;

II. for-loops with dynamic bounds are allowed in a program;

III. while-loops are allowed in a program.

In [21], the first relaxation has been considered: a novel automated procedure has been developed that derives a PPN from a class of affine nested loop programs called *Weakly Dynamic Programs* (WDPs). In this class of programs, the `if`-conditions might be dependent on some information that is unknown at compile-time and may change at run-time. In this dissertation, we have considered the other two more difficult relaxations. In Chapter 3, we considered relaxation II and presented a first approach for automated translation of affine nested loops programs with dynamic loop bounds (`Dynloop`) into input-output equivalent PPNs. Relaxation III, we considered in Chapter 4 by presenting a novel approach for automated translation of affine nested loop programs with while-loops (WLAPs) into input-output equivalent PPNs.

In contrast to deriving a PPN specification of a SANLP program, converting dynamic programs into PPNs in a systematic and automated way is a challenging and complex problem. This stems from the fact that the exact behavior of a dynamic program is unknown at compile-time. Therefore, for example, formal tools such as Exact Array Dataflow Analysis cannot be used to extract dependence relations in a dynamic program as this has been shown in Section 1.3. In Chapters 3 and 4, we demonstrated that although the exact behavior of dynamic programs with relaxations II and III is unknown at compile-time, still such programs can be analyzed and converted to an executable PPN specification in a systematic and automated way.

At a high-level, our approaches presented in Chapters 3 and 4 are similar and consist of three main steps. First, we extract an *approximated* dependency relation information from a dynamic program using the Fuzzy Array Dataflow Analysis (FADA) [37, 38] technique. We explain what approximation means. In Section 1.3 we demonstrated the difference of dependency patterns between dynamic and static programs. In static programs, different instances of a program correspond to one and the same single dependency pattern which is known at compile-time. In dynamic programs, data dependency patterns correspond to different *instances* of a dynamic program, and are unknown at compile time. This also means that the exact data dependency patterns in a dynamic program cannot be determined at compile-time. Therefore, we parameterize or *approximate* them using parameters whose values have to be set dynamically at run-time in order to preserve the initial data-flow in a program.

Second, we translate the initial program into dynamic Single Assignment Code (dSAC) [21] form and implement the general approach how to set the values of parameters introduced by FADA at run-time. A dSAC program is input-output equivalent to the corresponding dynamic program and it has the property that every data variable or an array element is written *at most once*. This implies that some variables may not be written at all. Also, at this step, the storage structure of the initial application is transformed such that each pair of statements communicates data over a dedicated multidimensional array.

Third, we demonstrate how the topology of the corresponding PPN and the code executed in each process are derived. Additionally, because the target model, Polyhedral Process Network (PPN), requires FIFO channels as communication medium, at this step memory array accesses are converted into managed dataflow over FIFO queues. Mapping such array communication onto FIFO channels requires complex address generators, especially if the arrays have multiple dimensions. Therefore, in Chapter 5, we addressed the Communication Model Identification (CMI) problem, which investigates communication characteristics of each Producer/Consumer (P/C) pair.

On the basis of the results obtained from Chapters 3,4,5,6 and experimental results, we can draw the following conclusions:

Conclusion I The work presented in this dissertation can be implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential dynamic programs. The *pn* [48] compiler which implements our approaches drew the attention of Intel Corporation and actually, Intel sponsored and evaluated these implementations.

Conclusion II With our approach that uses FADA analysis, we reveal all available task-level parallelism presented in the initial specification of a dynamic program. This allows for the utilization of multiple cores in an efficient way. This has been demonstrated in Chapter 6.

Conclusion III In some cases our parallelization approaches may exhibit some overhead introduced by creation of an excessive amount of control channels. This

will result in more run-time communication of control data in comparison to the control data communication in a PPN carefully optimized and derived by hand. Therefore, some optimization techniques have to be added to our approach in order to improve the quality of the generated PPNs in terms of communication control overhead. The investigation of such optimization techniques has already been started and sponsored by Intel Corporation. Moreover, some of the optimizations are under development and test in the current version of the *pn* [48] compiler.

Conclusion IV The control FIFO channels appear in a PPN derived from dynamic programs because the behavior of these programs is not completely known at compile-time. The presence of control FIFO channels introduces extra workload and communication overhead that are the consequences of the dynamic nature of the initial application. The analytical analysis conducted in Section 3.7 showed that the effect of these extra control structures and operations (overhead) on the performance of the PPN significanlty decreases when the granularity of the function calls executed inside the processes increases.

In the work presented in this dissertation, we considered a parallelization strategy that exploits task-level parallelism. Although, our approaches extract the maximum parallelism available, some other techniques can be used to extract other types of parallelism. For example, in future work, one can investigate on the transformations of PPNs similar to [20, 60], where a data-level partition strategy can be considered in order to achieve better execution performance and to automatically derive PPNs from dynamic applications.

Bibliography

[1] G. Martin. Overview of the MPSoC Design Challenge. In *Proc. DAC*, July 2006.

[2] Andrew Mihal and Kurt Keutzer. Mapping Concurrent Applications onto Architectural Platforms. In Axel Jantsch and Hannu Tenhunen, editors, *Networks on Chips*, pages 39–59. Kluwer Academic Publishers, 2003.

[3] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.

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

[5] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103, 1996.

[6] Sven Verdoolaege. Polyhedral process networks. Springer, 2010.

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

[8] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, DATE '04, pages 10340–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] E. A. de Kock. Multiprocessor mapping of process networks: a jpeg decoding case study. In *Proceedings of the 15th international symposium on System Synthesis*, ISSS '02, pages 68–73, New York, NY, USA, 2002. ACM.

[10] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. Networks on chip. chapter Guaranteeing the quality of services in networks on chip, pages 61–82. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[11] Basant Kumar Dwivedi, Anshul Kumar, and M. Balakrishnan. Automatic synthesis of system on chip multiprocessor architectures for process networks. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 60–65, New York, NY, USA, 2004. ACM.

[12] Jeronimo Castrillon, Ricardo Velasquez, Anastasia Stulova, Weihua Sheng, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 753–758, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[13] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In Andy D. Pimentel and Naehyuck Chang, editors, *ESTImedia*, pages 35–44. IEEE, 2009.

[14] Alexandru Turjan. Compiling nested loop programs to process networks, 2007. PhD thesis, Leiden University, The Netherlands.

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

[16] C.E Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1:36 – 47, 1954.

[17] Sven Verdoolaege. An integer set library for program analysis. *ACES symposium, Edegem, 7-8 september*, 2009.

[18] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[19] Paul Feautrier. Parametric Integer Programming. In *RAIRO Recherche Op'rationnelle, 22(3):243-268*, 1988.

[20] Sjoerd Meijer. *Transformations for Polyhedral Process Networks*. PhD thesis, Leiden University, The Netherlands, 2010.

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

[22] W.B. Pennebacker and J.L. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, 1993.

[23] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards; Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

[24] S. Arulampalam and S. Maskell. A Tutorial of Partical Filter for On-line Non-linear/Non-Gaussian Bayesian Tracking. *IEEE Trans. on Signal Processing*, pages 68–73, February 2002.

[25] Tie-Jun Shan and T. Kailath. Adaptive beamforming for coherent signals and interference. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33:527–536, 1985.

[26] Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. Translating Affine Nested-Loop Programs with Dynamic Loop Bounds into Polyhedral Process Networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010*, pages 21–30, Scottsdale, AZ, USA, October 2010.

[27] Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. Automated generation of polyhedral process networks from programs with dynamic loop bounds. *Accepted for publication in ACM Transactions on Embedded Computing Systems (TECS)*, 2012(1):xx–xx, 2012. Pre-print can be downloaded from `http://www.liacs.nl/~dmitryn/pb/TECS-2011-0228.R1.pdf`.

[28] Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. Automatic Translation of While-loop Affine Nested Loop Programs into Polyhedral Process Networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011*, Taipei, Taiwan, 2011.

[29] Dmitry Nadezhkin and Todor Stefanov. Identifying Communication Models in Process Networks Derived from Weakly Dynamic Programs. In *Proc. SAMOS X*, pages 372–379, July 2010.

[30] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000.

[31] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating Affine Nested-loop Programs to Process Networks. In *Proc. CASES'04*, Washington D.C., USA, September 23-25 2004.

[32] K. Knobe and V. Sarkar. Array SSA form and its use in Parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 107–120, San Diego, California, USA, January 1998.

[33] Paul Feautrier, Jean-Francois Collard, Michel Barreteau, Denis Barthou, Albert Cohen, and Vincent Lefebvre. The Interplay of Expansion and Scheduling in PAF. Technical report, PRiSM, University of Versailles, France, 1998. Report #1998/6.

[34] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Program.*, 23:191–219, April 1995.

[35] M. Griebl and J.-F. Collard. *Generation of Synchronous Code for Automatic Parallelization of while-loops*. EURO-PAR'95, Springer-Verlag LNCS, number 966, pp. 315-326, 1995.

[36] Martin Griebl and Christian Lengauer. A communication scheme for the distributed execution of loop nests with while loops. *Int. J. Parallel Programming*, 23, 1995.

[37] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 92–101, Santa Barbara, California, 1995. ACM Press.

[38] Denis Barthou Jean-Francois, Jean francois Collard, and Paul Feautrier. Fuzzy array dataflow analysis. In *Journal of Parallel and Distributed Computing*, pages 92–102, 1997.

[39] Paul Feautrier, Jean-François Collard, Michel Barreteau, Denis Barthou, Albert Cohen, and Vincent Lefebvre. The interplay of expansion and scheduling in paf, 1998.

[40] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proc. 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008.

[41] Martin Griebl and Christian Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum, 1996.

[42] Max Geigl, Martin Griebl, and Christian Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Comput.*, 25(12):1489–1510, 1999.

[43] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proc. International Conference on Compiler Construction (ETAPS CC'10)*, Paphos, Cyprus, 2010.

[44] Lawrence Rauchwerger and David Padua. Parallelizing while loops for multiprocessor systems. In *In Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[45] Tjerk Bijlsma, Marco J. G. Bekooij, and Gerard J .M. Smit. Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs. SAMOS'09, pages 140–148, 2009.

[46] Tjerk Bijlsma. *Automatic parallelization of nested loop programs for non-manifest real-time stream processing applications*. PhD thesis, Enschede, the Netherlands, July 2011.

[47] S. Geuns, T. Bijlsma, H. Corporaal, and M.J.G. Bekooij. Parallelization of While Loops in Nested Loop Programs for Shared-Memory Multiprocessor Systems. In *Proc. Int. Conf. Design, Automation and Test in Europe (DATE'11)"*, Grenoble, France, Mar 14–18 2011.

[48] pn compiler. `http://repo.or.cz/w/isa.git`.

[49] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Multi-dimensional incremental loop fusion for data locality. In *In Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pages 17–27, 2003.

[50] Sven Verdoolaege, Rachid Seghir, and Kristof Beyls. Analytical computation of ehrhart polynomials: Enabling more compiler analyses and optimizations. In *In CASES*, pages 248–258, 2004.

[51] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(8):983–996, August 2009.

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

[53] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14:403–420, 1970.

[54] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Realizations of the extended linearization model in the compaan tool chain. In *Proceedings of the 2nd Samos Workshop*, Samos, Greece, August 2002.

[55] Hristo Nikolov, Todor Stefanov, and Ed F. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(3):542–555, 2008.

[56] Tamas Farago. A framework for heterogeneous desktop parallel computing. Master's thesis, LERC, LIACS, 2009.

[57] Hristo Nikolov. *System-Level Design Methodology for Streaming Multi-Processor Embedded Systems*. PhD thesis, 2009. Leiden University, The Netherlands.

[58] Daedalus system-level design, http://daedalus.liacs.nl/.

[59] Alberto Sangiovanni-Vincentelli and Grant Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[60] Teddy Zhai, Hristo Nikolov, and Todor Stefanov. Mapping streaming applications considering alternative application specifications. *Accepted for publication in ACM Transactions on Embedded Computing Systems (TECS)*, 2012(1):xx–xx, 2012. Pre-print can be downloaded from `http://www.liacs.nl/~tzhai/pdf/Zhai-TECS2012.pdf`.

Index

Acknowledgments

Working on the Ph.D. was an incredible and often overwhelming experience. It was a time of professional and personal growth, a time of unforgettable experience living abroad and a time when new friends were met. It is my privilege and great pleasure to express my gratitude to those who have, directly or indirectly, supported me and helped me during my Ph.D. studies.

First of all, I would like to thank the former and the current members of the LERC group. It is an honor to say that I was a part of this group. The good advice, support and friendship of the LERC members has been invaluable on both an academic and a personal level, for which I am extremely grateful. My thoughts go out to Hristo Nikolov, Sven van Haastregt, Bin Jiang, Teddy Zhai, Mohammed al Hissi, Mohammed Bamakhrama, Emanuele Cannella and many others.

I also want to specially thank Sjoerd Meijer for being my "Dutch" friend and helping me to discover The Netherlands. Amongst the fruitful professional interactions at Niels Bohrweg, there always was time for jokes and joyful moments.

Furthermore, several professors outiside of my research group and/or Leiden University have contributed to building my way of thinking and knowledge at a level, giving me the confidence needed to finish Ph.D. research. In particular, Sven Verdoolaege, Andy Pimentel, Georgi Gaydadjiev, Andre Deutz, and many others. Thank you!

Also, I would like to acknowledge the financial, academic and technical support of the LIACS institute and its staff. I also thank the HR Department of Leiden University for their support and assistance during my Ph.D. studentship.

The work presented in this dissertation has been supported by the MEDEA+ NEVA project 2A703. I would like to thank the NEVA project for financially supporting my research.

I am very grateful to the committee members for the critical evaluation of my dissertation.

Especially, I would like to say “spasibo” to all my Russian friends I met in The Netherlands. Polina, Kate, Misha, Masha, Anton and many many others, thank you for being a small motherland for me. I sincerely appreciate our friendship and thousand of kilometers that now separate us will never fade my feelings to you. You are always welcome to visit me wherever I will be living.

I also want to thank those who put up with me throughout the whole Ph.D. process and shared with me their thoughts and experience: Luca, Alexander, Chris, Jorge, Marat, Niki, Oleg, Arseniy and many others.

In addition, I would like to thank my friend “ironman” Norman, whose self-discipline and musical talents have always been great inspirations for me.

At times, when I needed a cosy and homelike atmosphere, I often headed to Babbles restaurant. Thank you, Natali.

More than four years of my living in Leiden I spent at Mozartstraat. While staying there I had many unforgettable events and happy moments. Thank you, Oscar, for providing me this roof in Leiden. I will always remember this place as my home.

My special thoughts go to Sean who became my close friend and soul mate. I truly enjoyed the discussions and debates we had about almost any subject. Sometimes, we happen to look on things differently but had always respected and accepted each other’s opinion. Thank you and the best of luck in your personal pursuits.

With all my heart, I would like to thank Masha for her love and encouragement which supported me when I needed it the most.

Finally, I would love to express my gratitude to my mom, dad, and brother for their infinite support throughout everything.

Dmitry Nadezhkin
December, 2012
Leiden, The Netherlands

Samenvatting

Dit proefschrift concentreert zich op de automatische parallellisatie van sequentiele programma's met een zodanig adaptief en dynamisch gedrag dat zij geexecuteerd kunnen worden op een embedded systeem met meerdere processoren. In ons werk leiden wij Polyhedrale Proces Netwerken (PPN) specificaties af uit dynamische sequentiele programma's. Het sequentiele model matcht niet met de manier waarop multiprocessor systemen werken. Het PPN model is een deelverzameling van het Kahn Process Netwerk model, en staat veel dichter bij multiprocessor systemen. PPN kan met wiskundige methoden benaderd worden.

De methoden en technieken van dit proefschrift zijn belangrijke en niet-triviale uitbreidingen op voorafgaand werk over systematische en automatische afleiding van parallelle specificaties in de vorm van procesnetwerken, uit programma's die bestaan uit statische programmeerconstructies ("loops"). Deze programma's staan een automatische analyse toe en kunnen direct vertaald worden naar PPN's. De restrictie tot statische programma's geeft echter beperkte toepasbaarheid, programma's met adaptief en dynamisch gedrag passen niet binnen deze restrictie. Daarom proberen we in dit proefschrift deze beperkingen te verminderen.

Door het bestuderen van verschillende toepassingen besloten wij de volgende uitbreidingen van programma's te bekijken:

(1) dynamische if-condities worden toegestaan

(2) for-loops met dynamische grenzen worden toegestaan

(3) while-loops worden toegestaan.

Voor **(1)** hebben we een nieuwe methode ontwikkeld die werkt voor een klasse van affine loopprogramma's met dynamische if-condities. Deze condities kunnen afhankelijk zijn van informatie die niet bekend is op het tijdstip van de compilatie and die gedurende run-time aan verandering onderhevig kan zijn. We hebben ook gekeken naar de moeilijkere uitbreidingen **(2)** en **(3)**. In hoofdstuk 3 is een eerste stap gezet voor programma's met affine nested loops met dynamische grenzen. In

hoofdstuk 4, geven wij een nieuwe aanpak voor affine nested-loop programma's met while-loops.

In tegenstelling tot het afleiden van een PPN specificatie uit een statisch programma, is het systematisch en automatisch omzetten van dynamische programma's naar PPN's een complex en uitdagend probleem. Dit komt omdat de belangrijke stappen in het afleiden van een PPN programma dan niet werken. In hoofdstukken 3 en 4 laten we zien dat, ondanks dat het precieze gedrag van dynamische programma's niet bekend is op het tijdstip van de compilatie, het toch mogelijk is om dergelijke programma's te analyseren en te transformeren naar executeerbare PPN specificaties op een systematische en automatische manier.

In hoofdstuk 5 wordt een andere contributie beschreven. The PPN model gebruikt FIFO kanalen als communicatiemedium. Het gevolg is dat toegang tot het geheugen in het sequentiele programma geconverteerd moet worden naar dataflow over de FIFO kanalen. Hoofdstuk 5 laat zien hoe dit gedaan kan worden: communicatie karakteristieken tussen twee communicerende processen in PPN worden bestudeerd.

De methoden en technieken die in dit proefschrift gepresenteerd worden, hebben geresulteerd in uitbreidingen van de pn compiler (http://daedalus.liacs.nl). In hoofdstuk 6 wordt de parallellisatieaanpak van hoofdstuk 3 geevalueerd op de toepassing "Low Speed Obstacle Detection".

Curriculum Vitae

Dmitry Nadezhkin was born on 15^{th} of July, 1981 in Arzamas-16, USSR. In 2003, he received his Master Degree in Mathematics from the Lomonosov Moscow State University. During his M.Sc. study, Dmitry Nadezhkin worked at designing software targeting high-performance computing systems. In 2006, Dmitry joined the Leiden Embedded Research Center (LERC) which is part of the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University where he was appointed as a research assistant (Ph.D. student). He was involved in the NEVA project which deals with Networks on Chips Design Driven by Video and Distribution Applications, and conducted research in the area of automatic parallelization of program code with dynamic constraints. As a part of his work, he developed a FIFO library that allows to run Polyhedral Process Networks on the IBM Cell multiprocessor platform. The research work culminated in the writing of this Ph.D. dissertation in 2012.

List of Publications

- Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov, "Automated Generation of Polyhedral Process Networks from Programs with Dynamic Loop Bounds", Accepted for publication in ACM Transactions on Embedded Computing Systems (TECS), vol. x, Issue x, pp. xx - xx, June 2012. Pre-print of the paper can be downloaded from `http://www.liacs.nl/~dmitryn/pb/TECS-2011-0228.R1.pdf`

- Dmitry Nadezhkin, Todor Stefanov: "Automatic Translation of While-loop Affine Nested Loop Programs into Polyhedral Process Networks", In Proc. 9th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'11), Taipei, Taiwan, October 13-14, 2011.

- Dmitry Nadezhkin, Hristo Nikolov, Todor Stefanov: "Translating Affine Nested Loop Programs with Dynamic Loop Bounds into Polyhedral Process Networks" In Proc. 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'10), Scottsdale, Arizona USA, October 24-29, 2010. **WINNER of the 2010 ESTIMedia Best Paper Award!**

- Dmitry Nadezhkin, Todor Stefanov: "Identifying Communication Models in Process Networks derived from Weakly Dynamic Programs", In Proc. "10th Int. Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS'10)", pp. 372-379, Samos, Greece, July 19-22, 2010

- Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, Ed F. Deprettere, "Realizing FIFO Communication When Mapping Kahn Process Networks onto the Cell" In Proc. "9th Int. Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS'9)", pp. 308-317, Samos, Greece, July, 2009

- Sjoerd Meijer, Sven van Haastregt, Bart Kienhuis, Dmitry Nadezhkin: "Kahn Process Network IR Modeling for Multi core Compilation", Technical Report, Leiden University, 2008