Translating Affine Nested-Loop Programs with Dynamic Loop Bounds into Polyhedral Process Networks

Dmitry Nadezhkin Hristo Nikolov Todor Stefanov
Leiden Institue of Advanced Computer Science, Leiden University, The Netherlands
{dmitryn, nikolov, stefanov}@liacs.nl

Abstract*—The Process Network (PN) is a suitable parallel model of computation (MoC) used to specify embedded streaming applications in a parallel form facilitating the efficient mapping onto embedded parallel execution platforms. Unfortunately, specifying an application using a parallel MoC is very difficult and highly error-prone task. To overcome the associated difficulties, an automated procedure exists for derivation of a specific polyhedral process networks (PPN) from static affine nested loop programs (SANLPs). This procedure is implemented in the *pn* complier. However, there are many applications, e.g., multimedia applications (MPEG coders/decoders, smart cameras, etc.) that have adaptive and dynamic behavior which can not be expressed as SANLPs. Therefore, in order to handle more dynamic multimedia applications, in this paper we address the important question whether we can relax some of the restrictions of the SANLPs while keeping the ability to perform compile-time analysis and to derive PPNs. Achieving this would significantly extend the range of applications that can be parallelized in an automated way. The main contribution of this paper is a first approach for automated translation of affine nested loops programs with dynamic loop bounds into input-output equivalent polyhedral process networks.

I. Introduction

Moving from sequential computing to parallel computing has become necessary nowadays because single-processor embedded systems can not cope anymore with applications complexity, throughput, and power consumption constraints that are inherent to so many embedded applications. 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 programmed in a way that the available parallelism is revealed and exploited efficiently. However, 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 building 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 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 *C* 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.

```
parameter N 10 100;

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

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

Fig. 1: Pseudo code of a SANLP.

An example of a SANLP is given in Figure 1. A SANLP 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 [4]. The SANLPs can be converted in an automated way into Polyhedral Process Networks [3].

The target Polyhedral Process Networks (PPNs) [5] is a special (static) case of the Kahn Process Networks (KPNs) [6] 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 dataflow 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 [7], [8], [9], [10], [11], [12].

Many scientific, matrix computation, and signal processing applications can be specified as static affine nested loop programs, and therefore, the *pn* compiler [3] can be used to derive equivalent parallel PPN specifications. However, many multimedia applications such as MPEG coders/decoders, Smart Cameras, etc. have adaptive and dynamic behavior which can not be expressed as static affine nested loop programs. In order to handle dynamic applications, in this paper we address the important question whether we can relax some of the restrictions of the SANLPs 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. The main contribution of this paper is a first approach for automated translation of affine nested loop programs with dynamic loop bounds into input-output equivalent polyhedral process networks.

A. Motivating example

As a motivating example, we use an application from the smart cameras domain called low speed obstacle detection (LSOD). With the LSOD description below, we illustrate a program that has the specific dynamic behavior we consider in this paper and we outline the problems introduced by this behavior.

The LSOD application 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 [13] 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 2. Applying different image processing algorithms on these images, hypotheses whether

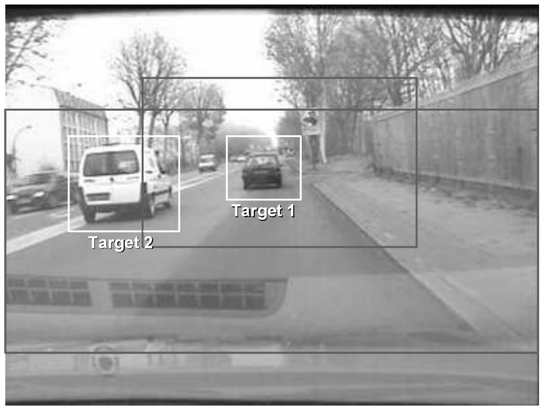


Fig. 2: LSOD applied on real data. The vehicles in front of the camera are detected and tracked. The dark rectangles depict the area of the image that is processed.

```
1 for k = 1 to Targets,

2   [Height,Width] = getLSODTarget()

3   for j = 0 to Height+1,
4     for i = 0 to Width+1,
5       img[j,i] = readTarget()
6     endfor
7   endfor

8   for j = 1 to Height,
9     for i = 1 to Width,
10      img_out[j-1,i-1] = 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-1,i-1] = absVal( img_out[j-1,i-1] )
12    endfor
13  endfor

14  for j = 0 to Height-1,
15    for i = 0 to Width-1,
16      vsum[i] = vertSum( vsum[i], img_out[j,i] )
17    endfor
18  endfor

19 endfor
```

Fig. 3: Pseudo code of the edge detection part of the motivating example. Target size is specified by variables Height and Width. For the sake of brevity, the information about targets position is omitted.

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 2, 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.

The edge detection part of the LSOD application, shown in Figure 3, 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 and dimensions 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 not known at compile-time, and therefore, the *pn* compiler [3] cannot handle the program shown in Figure 3. In this paper we propose a solution approach to this problem by introducing a novel procedure for automated translation of affine nested loops programs with dynamic loop bounds into input-output equivalent polyhedral process networks.

The remaining part of the paper is organized as follows. In the following section, we introduce some notations and present two techniques currently used to analyze sequential programs. This is needed for better understanding of the solution approach we propose and discuss in Section III. An application of our solution approach to the program in Figure 3 is presented in Section IV. Section V covers the related works. Finally, Section VI concludes the paper.

II. Background

In this section, we introduce some notations used throughout the paper. Also, for better understanding of the solution approach, we briefly present two state-of-the-art techniques used to analyze sequential programs. The first one, called Exact Array Dataflow Analysis (EADA) [14], is used to analyze static programs, namely SANLPs. EADA is implemented in the *pn* compiler for the translation of SANLPs to polyhedral process networks. We formally describe EADA in Section II-B. The second technique, which we present in Section II-C, allows for the analysis of programs with more relaxed constraints than SANLPs. That is, we consider the Fuzzy Array Dataflow Analysis (FADA) introduced in [15]. FADA is an enhanced version of EADA and it is used in [16] to analyze Weakly Dynamic Programs (WDP). WDPs are class of affine nested loop programs which may have *if*-conditions dependent on data which is unknown at compile-time and which may change at run-time. In [16], FADA is used for translation of WDPs to equivalent PNs. Similarly to SANLPs, in WDPs loop bounds have to be affine functions of enclosing loop iterators and static parameters. In this paper, we further relax these restrictions by considering sequential affine nested loop programs with dynamic loop bounds. In this section, we introduce FADA because an important part of the solution approach presented in Section III is based on this technique.

A. Notations

An iteration vector x of a statement is built of iterators of surrounding loops. The set of values of an iteration vector for which a statement is executed represents an iteration domain, denoted by $\mathbf{D}()$. For example, the iteration domain of statement $S2$ in Figure 4(a) is: $\mathbf{D}(S2) = \{1 \leq i \leq N \wedge i \leq j \leq M \wedge j \leq 2\}$. An evaluation of a single statement W on iteration x is called an operation and denoted as $\langle W, x\rangle$. By "$\prec$" we denote ordering of operations. An operation $\langle W, x\rangle$ is evaluated before an operation $\langle R, y\rangle$ ($\langle W, x\rangle \prec \langle R, y\rangle$) according to the program sequence if: 1) x lexicographically precedes y; or 2) if $x = y$ and statement W precedes statement R in the program text. As described in [14], order "$\prec$" can be expanded to a system of linear inequalities. With "max" we denote the lexicographical maximum operator. In this paper, we use `Dynloop` to designate affine nested loop programs with dynamic loop bounds.

```
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,
8      if  j <= 2 then
S2:        y[j] = F2()
10     endif
S3:      [] = F3(y[j])
12   endfor
13 endfor
```

(a) Static Affine Nested Loop Program

```
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,
8      if  y[j] <= 2 then
S2:        y[j] = F2()
10     endif
S3:      [] = F3(y[j])
12   endfor
13 endfor
```

(b) Weakly Dynamic Program

Fig. 4: Examples of SANLP and WDP programs. The only difference is that in WDP, the conditional statement in line 8 is data-dependent.

B. Exact Array Dataflow Analysis

In this section we formally describe the EADA algorithm, which is used to perform the dependence analysis on static programs. 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 4(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 $\langle W, x\rangle$ and $\langle R, y\rangle$, where the first operation writes to an array and the second operation reads from it. To find whether the operation $\langle W, x\rangle$ is a source for operation $\langle R, y\rangle$, we need to build and solve a system of linear inequalities:

$$\mathbf{Q}_{WR}(y) = \{x \mid \begin{array}{ll} x \in \mathbf{D}(W), & \text{(c1)} \\ \mathcal{I}_W(x) = \mathcal{I}_R(y), & \text{(c2)} \\ \langle W, x\rangle \prec \langle R, y\rangle. & \text{(c3)} \end{array} \tag{1}$$

The first constraint (c1) states that the source iteration x has to exist, i.e., it has to belong to the iteration domain of a W statement. 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 $\langle W, x\rangle$ uses an affine indexing function $\mathcal{I}_W()$ and operation $\langle R, y\rangle$ uses an affine indexing function $\mathcal{I}_R()$. The (c3) constraint determines an order of operations, i.e., source operation $\langle W, x\rangle$ has to be evaluated *before* operation $\langle R, y\rangle$.

There might be many operations of a single statement satisfying system (1), i.e., writing to the same array element. However, we are interested in the last write operation before reading by $\langle R, y\rangle$ from the same element occurs. Therefore, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(y)$:

$$\mathbf{K}_{WR}(y) = \max \ \mathbf{Q}_{WR}(y). \tag{2}$$

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, we have to consider all pairs $W_1/R,\ldots W_m/R$. The actual source is the "last" operation between all operations of all statements:

$$\sigma(\langle R,y\rangle) = \max\ \{\langle W_k, \mathbf{K}_{W_kR}(y)\rangle \ \mid \ \ k \in [1,m]\}. \tag{3}$$

For example, consider the program in Figure 4(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 (1) as depicted in Table I (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 $S3$.

$\mathbf{Q}_{\text{S1S3}}((i_3,j_3))$	$\mathbf{Q}_{\text{S2S3}}((i_3,j_3))$	
$1 \le k \le M$	$1 \le i_2 \le N \wedge i_2 \le j_2 \le M \wedge$ $j_2 \le 2$	(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 I: Examples of system (1) for S1S3 and S2S3 statements.

After solving the parametric integer linear problems (PILPs) formulated in Table I and finding "max" according to Equation 3, the source operation $\sigma(\langle \text{S3},(i_3,j_3)\rangle)$ for the data read by statement S3 is:

$$\begin{array}{lll}\textbf{if} \quad j_3 \le 2 & \textbf{then} & \langle S2,(i_3,j_3)\rangle \\ & \textbf{else} & \langle S1,(j_3)\rangle.\end{array} \tag{4}$$

The solution above shows that the source of the data for statement $S3$ of the program in Figure 4(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 $S2$.

C. Fuzzy Array Dataflow Analysis

In this section, we formally describe the Fuzzy Array Dataflow Analysis (FADA). The FADA algorithm is used to perform dependence analysis on Weakly Dynamic Programs (WDP) which have data-dependent *if*-conditions [16]. We introduce FADA because it is an important part of the solution we present in Section III.

Consider two statement W and R of a *weakly dynamic* program. Operation $\langle W,x\rangle$ writes to and operation $\langle R,y\rangle$ reads from the same array. Moreover, let statement W be surrounded by a data-dependent *if*-condition. As a running example, consider Figure 4(b): statements $S2$ and $S3$ are W and R, respectively, and the *if*-condition in line 8 surrounding statement $S2$ is a data-dependent condition.

In Section II-B, we showed that in order to have two operations $\langle W,x\rangle$ and $\langle R,y\rangle$ of a *static* program dependent, they have to comply to the system of linear inequalities (1). In the same way, to find whether operation $\langle W,x\rangle$ is a source for operation $\langle R,y\rangle$ in a *dynamic* program, we need to build and solve a system of linear inequalities:

$$\mathbf{Q}_{WR}(y,\alpha) = \{x \ \mid \ \begin{array}{ll} x \in \mathbf{D}(W), x = \alpha, & (c1) \\ \mathcal{I}_S(x) = \mathcal{I}_R(y), & (c2) \\ \langle S,x\rangle \prec \langle R,y\rangle. & (c3)\end{array} \tag{5}$$

The meaning of constraints $(c2)$ and $(c3)$ are the same as in system (1): 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 $\langle R,y\rangle$ 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 $\langle W,x\rangle$ may occur. We do not know exactly at which iteration points $x \in \mathbf{D}(W)$ writing to the array occurs, but we assume that this happens for iterations $x = \alpha$, where α is a free parameter which values have to be determined at run-time. Because source operations satisfying system (5) are not exact, we call them *approximated* sources.

Similarly to the EADA algorithm, we are interested in the last write operation before reading by $\langle R,y\rangle$ from the same element occurs, i.e., the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(y,\alpha)$:

$$\mathbf{K}_{WR}(y,\alpha) = \max \mathbf{Q}_{WR}(y,\alpha). \tag{6}$$

Finally, we need to consider all statements $W_1,\ldots,W_m$ writing to the same array element. For each W_k, $k = [1..m]$, we find approximated source. To find the source, we combine all approximated sources as described in [15]:

$$\sigma(\langle R,y\rangle,\alpha) = \max\{\langle W_k, \mathbf{K}_{W_kR}(y)\rangle \mid \ \ k \in [1,m]\}. \tag{7}$$

For example, consider the WDP depicted in Figure 4(b). There are two statements $S1$ and $S2$ writing to array y and one statement $S3$ which reads from it. For every pair $S1S3$ and $S2S3$ we build the systems of linear inequalities (5) which is depicted in Table II. 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 II). However, for pair $S2S3$ (see system $\mathbf{Q}_{S2S3}((i_3,j_3),(\alpha_i,\alpha_j))$), we introduce parameters α_i and α_j as shown in system (5), because statement $S2$ is surrounded by the dynamic *if*-condition at line 8 in Figure 4(b) 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.

$\mathbf{Q}_{S1S3}((i_3,j_3))$	$\mathbf{Q}_{S2S3}((i_3,j_3),(\alpha_i,\alpha_j))$	
$1 \le k \le M$	$1 \le i_2 \le N \wedge i_2 \le j_2 \le M \wedge$ $i_2 = \alpha_i \wedge j_2 = \alpha_j$	(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 II: Examples of system (5) for S1S3 and S2S3 statements.

Approximated sources in S1S3 and S2S3 pairs are found by solving the parametric integer linear problems (PILPs) formulated in Table II. The "max" source defined in Equation 7 is determined by recurrent algorithm of combining direct dependencies described in Section 5.2 of [15]. Thus, the source operation for statement S3: $\sigma(\langle S3,(i_3,j_3)\rangle,(\alpha_i,\alpha_j))$ is:

$$\begin{array}{lll}\textbf{if} \quad i_3 \ge \alpha_i \wedge j_3 == \alpha_j & \textbf{then} & \langle S2,(\alpha_i,\alpha_j)\rangle \\ & \textbf{else} & \langle S1,(j_3)\rangle.\end{array} \tag{8}$$

From Solution 8 above, we see 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

```
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] )
```

Fig. 5: An example of a `Dynloop` program.

previous evaluations of the condition at line 8 in Figure 4(b) 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 4, Solution 8 is approximated, because it depends on parameters (α_i, α_j) that are determined at run-time.

III. Solution approach

In this section, we present the compile-time procedure we have devised for translating affine nested loop programs with dynamic loop bounds (`Dynloop`) into input-output equivalent polyhedral process networks (PPN). We have found out that a `Dynloop` program can be formally represented as a Weakly Dynamic Program (WDP). Therefore, in the proposed solution approach we can employ the FADA dependence analysis technique, described in Section II-C.

The procedure for translating `Dynloop` programs to polyhedral process networks consists of 4 steps. First, the initial `Dynloop` program is represented as a WDP. Second, we find all data dependency in the corresponding WDP program by applying the FADA analysis on it. Recall that the result of the analysis is approximated, i.e., it depends on some parameters which values are determined at run-time. Third, based on the results of the analysis, we create a *dynamic* Single Assignment Code (dSAC) representation of the WDP program. The dSAC was proposed in [16] as an extension of the SAC [14]. A dSAC program is input-output equivalent to the corresponding WDP 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 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 variables. The generation of the control variables has been studied in [16], whereas, in this section, we present an extension to this procedure. In the last forth step, the topology of the corresponding PPN is derived, as well as the code executed in each process. In the remaining part of this section, we consider the four steps in more detail and we also illustrate the solution approach using the example shown in Figure 5.

Step 1 (Dynloop-to-WDP)

Consider the `Dynloop` program in Figure 5. 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. 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 6(a). Such modification is general and applicable to any `Dynloop` program. First,

```
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

Fig. 6: A WDP program equivalent to the `Dynloop` program in Figure 5 and its corresponding dSAC.

we substitute the upper bound of the loop at line 3 in Figure 5 with a constant equal to the maximum value of $f()$, denoted by `max_f`, see line 4 in Figure 6(a). The value of `max_f` can be determined by studying the range of function $f()$. If it is not possible, we can use the following approach. Assume that the capacity of the array `y` is 100 elements. Then, by taking into account the array indexing function at line 4 of Figure 5 and that the program is correct, we can deduce that the maximum value of iterator *i* and, consequently the `max_f` equals to 100. In addition, 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 6(a), just before the *for*-loop. The same array elements are used in evaluating the *if*-condition at line 5 in Figure 6(a), which preserves the original program behavior. This newly created program belongs to the class of the *weakly dynamic* programs. Since the loop bounds of the program in Figure 6(a) are fixed and known at compile-time, we can apply the FADA algorithm on this program to perform dependence analysis.

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

Step 2 (FADA analysis)

The WDP program in Figure 6(a) has two statements $S1$ and $S2$ which communicate through array `y`. According to FADA, for the pair $S1S2$, we build the system of linear inequalities shown in Table III which corresponds to Equation 5. Constraint $c1$ in Table III describes all possible source iterations of statement $S1$, i.e., its iteration domain. Parameters (α_j, α_i) store the iteration point (j_1, i_1) of statement $S1$ where writing to array `y` may occur.

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

TABLE III: An example of system (5) for S1S2 pair.

The approximated source operation defined in Equation 7 for statement $S2$: $\sigma(\langle S2, ()\rangle, (\alpha_j, \alpha_i))$ is:

$$\begin{array}{llll}\textbf{if} & N \geq \alpha_j \wedge 5 == \alpha_i & \textbf{then} & \langle S1, (\alpha_j, \alpha_i)\rangle \\ & & \textbf{else} & \perp .\end{array} \tag{9}$$

From Solution 9 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 6(a) is evaluated to `true`, then the source is statement $S1$. Otherwise, the source for `y[5]` is not statement $S1$. If this is the case, 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.

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 for our running example is shown in Figure 6(b) which is in a dynamic single-assignment-code (dSAC) form. The dSAC is an extension of the SAC [14]. 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 9, we modify the WDP in Figure 6(a) and generate the dSAC in Figure 6(b) by inserting the code lines 10-14 shown in Figure 6(b). This code is needed to implement array element accesses such that the dependences identified by FADA are respected. The *if*-condition at line 10 implements the condition that the FADA analysis gave us in Solution 9. Recall that when the *if*-condition evaluates to true, then the source of the data is statement $S1$. This is captured by 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, the FADA algorithm introduces parameters. In our example, there are two parameters (see line 10 in Figure 6(b)) which are reflected in the following way. Parameter $c1$ corresponds to α_j. It is related to iterator j and may have values $c1 \in [1..\mathtt{N}]$. Parameter $c2$ corresponds to α_i. It is related to iterator i and may have values $c2 \in [1..\mathtt{max\_f}]$. 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's behavior.

Step 3 (Control variables)

In order to keep the functionality of the dSAC equivalent to the functionality of the initial WDP, we introduce control variables used to propagate parameter values at run-time. That is, an array of control variables is added for every parameter introduced by FADA. A control variable is used to store a parameter value for every iteration. For our example in Figure 6(b), two new control variables `ctrl_c1` and `ctrl_c2` are introduced in the program as shown in Figure 7(a). They correspond to parameters $c1$ and $c2$ derived by FADA.

We use the original index function used with the data variable `y`, i.e., `y[i]`, to perform the access to the control variables, i.e., `ctrl_c1[i]` and `ctrl_c2[i]`. The control variables 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 control variables are initialized as follows:

$$\forall \mathtt{i} : 1 \leq \mathtt{i} \leq \mathtt{max\_f} : \quad \begin{array}{l}\mathtt{ctrl\_c1[i]} = \mathtt{N} + 1, \\ \mathtt{ctrl\_c2[i]} = \mathtt{max\_f} + 1.\end{array} \tag{10}$$

This initialization is not shown in Figure 7(a) for the sake of brevity. Writing to the control variables is performed just after function $F1()$, see lines 7 and 8 in Figure 7(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 parameters $c1$ and $c2$ at lines 12 and 13. These parameters are used to evaluate the conditions at line 14 which determine the source of the data for function $F2()$. With the introduction of the control variables to the program shown in Figure 7(a), this program is input-output equivalent to the program in Figure 6(a).

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 variables `ctrl_c1[i]` and `ctrl_c2[i]` that initialize parameters $c1$ and $c2$ at lines 12 and 13 in Figure 7(a) are not in a single assignment form, i.e., `ctrl_c1[i]` and `ctrl_c1[i]` may be written more than once. Therefore, the program in Figure 7(a) is not a dSAC program. 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 must be in a single assignment form. A novel contribution of our work is a procedure that extends the control variables generation described in the previous section. Our extension solves the problem that the control variables in Figure 7(a) are not in the single assignment form. Below, we explain how such control variables are transformed into a single assignment form.

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

TABLE IV: ILP system (5) for the control variables at lines 10, 11, 14 and 15.

In order to represent the program in Figure 7(a) 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 the control variables occur inside a block surrounded by a dynamic *if*-condition. We achieve this in the following way. While keeping the same functionality, we introduce additional control variables `ctrl_c1_` and `ctrl_c1_` *outside* the block surrounded by the dynamic *if*-condition, see lines 10,11,14 and 15 in Figure 7(b). This program is input-output equivalent to the program in Figure 7(a). The new control variables are written (lines 10 and 11) at every iteration of the *for*-loops and take the same values as control variables `ctrl_c1` and `ctrl_c2`. On these new control variables `ctrl_c1_` and `ctrl_c1_`, we can perform the *static* exact array dataflow 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],
6        y_1[j,i] = F1()
7        ctrl_c1[i] = j
8        ctrl_c2[i] = i
9      endif
10   endfor
11 endfor

12 c1 = ctrl_c1[5]
13 c2 = ctrl_c2[5]
14 if c1 <= N & c2 == 5,
16   in_0 = y_1[c1,c2]
17 else
18   in_0 = 0
19 endif
20 [] = F2( in_0 )
```

(a) Initial dSAC shown in Figure 6(b) with control variables

```
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        ctrl_c1[i] = j
8        ctrl_c2[i] = i
9      endif
S1:      ctrl_c1_[i] = ctrl_c1[i]
S1:      ctrl_c2_[i] = ctrl_c2[i]
12   endfor
13 endfor

S2: c1 = ctrl_c1_[5]
S2: c2 = ctrl_c2_[5]
16 if c1 <= N & c2 == 5,
18   in_0 = y_1[c1,c2]
19 else
20   in_0 = 0
21 endif
22 [] = F2( in_0 )
```

(b) Modified dSAC code with new control variables

```
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        ctrl_c1[i] = j
8        ctrl_c2[i] = i
9      endif
10     ctrl_c1_1[j,i] = ctrl_c1[i]
11     ctrl_c2_1[j,i] = ctrl_c2[i]
12   endfor
13 endfor

14 if max_f >= 5,
15   c1 = ctrl_c1_1[N, 5]
16   c2 = ctrl_c2_1[N, 5]
17 else
18   c1 = N + 1
19   c2 = max_f + 1
20 endif
21 if c1 <= N & c2 == 5,
22   in_0 = y_1[c1,c2]
23 else
24   in_0 = 0
25 endif
26 [] = F2( in_0 )
```

(c) Final dSAC

Fig. 7: Examples of the initial dSAC with control variables, the modified dSAC and the final dSAC.

presented in Section II-B. We can always do this, because the new control variables are not surrounded by the dynamic *if*-condition.

As it has been shown in Section II-B, we need to build a system of linear inequalities, which for the pairs S1S2 in Figure 7(b) is shown in Table IV. The system is build for each pair $S1S2$, i.e., $S1$ at line 10 with $S2$ at line 14 and $S1$ at line 11 with $S2$ at line 15. After solving the system and finding the maximum according to Equation (3), the final solution and the source operation is:

$$\begin{array}{llll} \textbf{if} & \texttt{max\_f} \geq 5 & \textbf{then} & \langle S1, (N, 5)\rangle \\ & & \textbf{else} & \perp . \end{array} \quad (11)$$

We use this solution to replace the original one-dimensional arrays `ctrl_c1_` and `ctrl_c2_`, see lines 10,11,14 and 15 in Figure 7(b), with two-dimensional arrays `ctrl_c1_1` and `ctrl_c2_1` shown at lines 10, 11, 15, and 16 in Figure 7(c). The *if*-condition at line 14 in Figure 7(c) implements the *if*-condition of Solution 11. If it evaluates to `true`, then the source of the parameters $c1$ and $c2$ is `ctrl_c1_1[N,5]` and `ctrl_c1_2[N,5]`, respectively (see lines 15 and 16). In case the condition evaluates to `false`, then the initial values of the control variables will be used. Recall that in our approach, the control variables used to propagate parameter values are initially set as shown in Equation 10. Therefore at lines 18 and 19 in Figure 7(c), we put assignments $c1 = \texttt{N}+1$ and $c2 = \texttt{max\_f} + 1$, respectively.

The program in Figure 7(c) is in a dSAC form because the new control variables `ctrl_c1_1` and `ctrl_c1_2` used to initialize parameters $c1$ and $c2$ are 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 5. We use this dSAC to generate a process network which is explained in the next section.

Step 4 (dSAC-to-PPN synthesis)

The last step of the procedure to translate `Dynloop` programs into equivalent polyhedral process networks (PPN) is the dSAC-to-PPN synthesis step. Recall that a PPN consists of autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels. We describe how the processes and FIFO channels are created from the corresponding dSAC program.

The procedure of PPN synthesis consists of 3 steps. First, based on the dSAC representation of a `Dynloop` program, the topology of the PPN is created. The topology is formed by instantiating processes and communication channels. It is important to note, that in this step, the created communication channels are not FIFOs but multi-dimensional arrays. Second, internal code structure of each process is derived from the dSAC specification. Third, the multi-dimensional arrays that are used for data communication between function statements in the dSAC are replaced by FIFO channels. Also, 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. Below, we explain the three steps in more detail using the dSAC in Figure 7(c).

Topology of a PPN

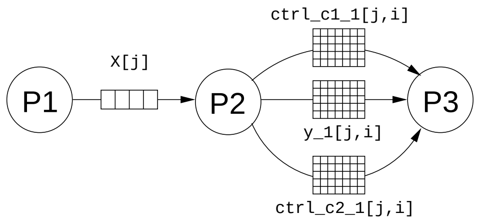
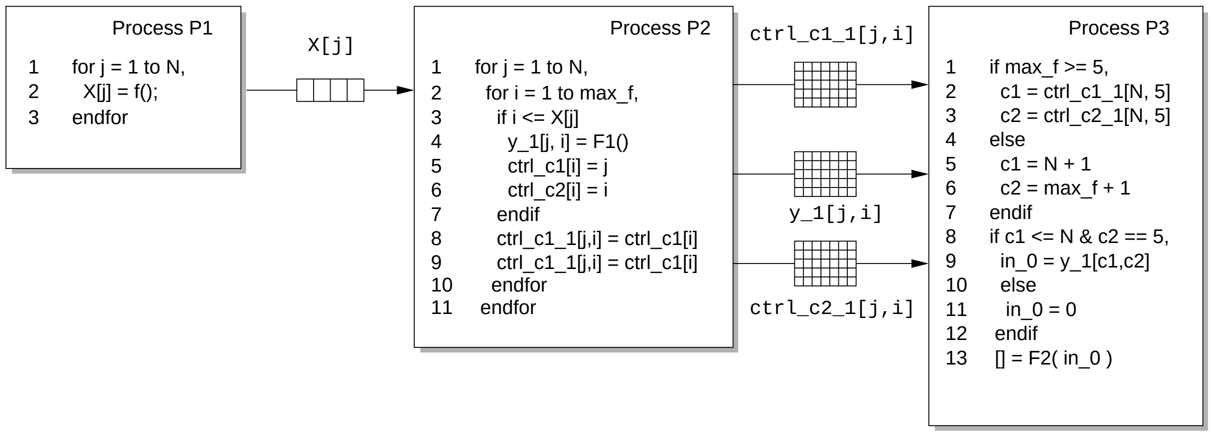


Fig. 9: The topology of the PN derived from the dSAC in Figure 7(c).

The PPN corresponding to the dSAC in Figure 7(c) is shown in Figure 9. This PPN consists of 3 processes and 4 communication channels. We explain how these processes and communication channels are created. In our approach, a process is created for



```
Process P1
1     for j = 1 to N,
2       X[j] = f();
3     endfor
```

```
Process P2
1     for j = 1 to N,
2       for i = 1 to max_f,
3         if i <= X[j]
4           y_1[j, i] = F1()
5           ctrl_c1[i] = j
6           ctrl_c2[i] = i
7         endif
8         ctrl_c1_1[j,i] = ctrl_c1[i]
9         ctrl_c1_1[j,i] = ctrl_c1[i]
10      endfor
11    endfor
```

```
Process P3
1     if max_f >= 5,
2       c1 = ctrl_c1_1[N, 5]
3       c2 = ctrl_c2_1[N, 5]
4     else
5       c1 = N + 1
6       c2 = max_f + 1
7     endif
8     if c1 <= N & c2 == 5,
9       in_0 = y_1[c1,c2]
10      else
11       in_0 = 0
12     endif
13     [] = F2( in_0 )
```

Fig. 8: The internal code structure of each process in the PPN derived from the dSAC in Figure 7(c).

every function statement in the dSAC program. Therefore, the PPN in Figure 9 has three processes: process $P1$ corresponds to function $f()$ at line 3 in Figure 7(c), process $P2$ corresponds to function $F1()$ at line 6, and process $P3$ corresponds to $F2()$ at line 26 in the same Figure. The four communication channels correspond to arrays which are in a single assignment form in the dSAC in Figure 7(c). These arrays are: one-dimensional array `X[j]` at line 3 and 5 in Figure 7(c), two-dimensional data array `y_1[j,i]` at lines 6 and 22, and 2 two-dimensional control variable arrays `ctrl_c1_1[j,i]` and `ctrl_c2_1[j,i]` at lines 10, 11, 15 and 16 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 a single assignment form of array `y`. The latter array is used in the initial program shown in Figure 5 after application of the FADA analysis on the WDP program in Figure 6(a) as described in Step 2 of our solution approach. The control variables arrays `ctrl_c1_1[j,i]` and `ctrl_c2_1[j,i]` are introduced and transformed in a single assignment form in Step 3 of our solution approach. In the following step, we describe how the internal code structure of each process is created.

Internal code structure

Let us consider Figure 8, where the internal code structures of all processes of the PPN in Figure 9 are depicted. Below we explain how the internal code structure of each process is derived from the corresponding dSAC specification shown in Figure 7(c).

Every process executes a sequential nested loop program, derived from the dSAC program. The internal code structure of each process is formed by code pieces of the dSAC code. Based on the example of the dSAC shown in Figure 7(c), the internal code for processes $P1$, $P2$ and $P3$ is derived as follows. The internal code structure of process P1 depicted in Figure 8 is formed by lines 2, 3 and 13 of the dSAC code in Figure 7(c). The internal code structure of process P2 is formed by lines 2, 4–13 of the same dSAC specification. And the internal code structure of process P3 is formed by lines 14–26. In the following step, we explain how the multi-dimensional arrays are replaced with FIFO channels and synchronization between processes is realized with FIFOs. This process is called *Linearization*.

Linearization

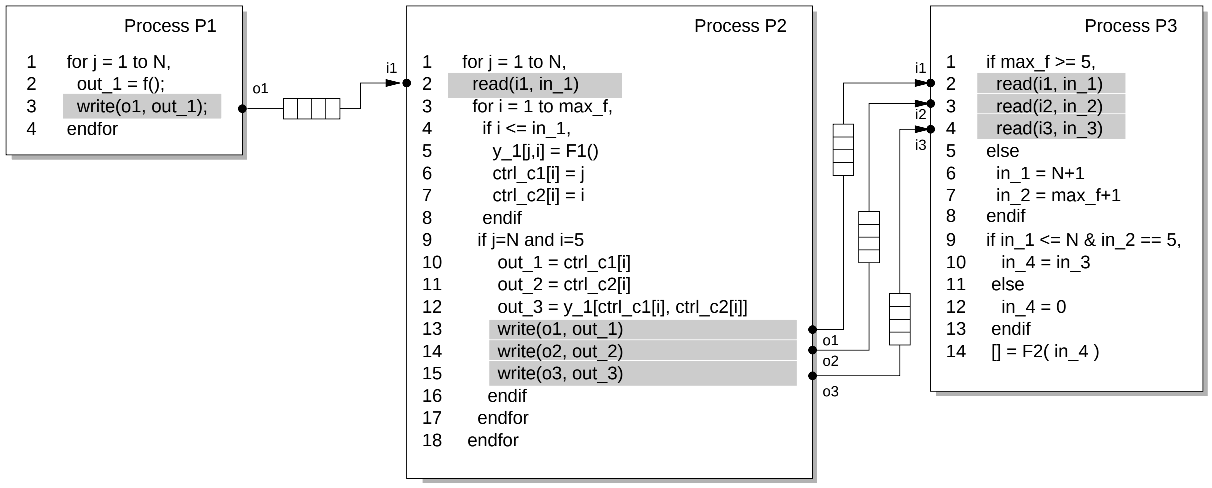
In the PPN depicted in Figure 8, processes are connected with communication channels which are the multi-dimensional arrays used in the dSAC shown in Figure 7(c). However, the processes in our PPNs synchronize using a blocking read/write on an empty/full FIFO channel, i.e., an execution of a process 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.

To implement the Linearization, we adapted the approaches proposed in [17], [18]. 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 accesses. The result of the linearization applied on the dSAC in Figure 7(c) is shown in Figure 10. In each process, the multi-dimensional arrays accesses are substituted by reading/writing primitives from/to FIFO channels. Internally, these read/write primitives realize synchronization between processes. For example, writing to the control variables array at lines 8 and 9 of process $P2$ in Figure 8 are substituted by writing to the FIFOs at lines 13 and 14 in process $P2$ in Figure 10.

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 3 channels via ports $o1$, $o2$ and $o3$ at lines 13, 14 and 15, respectively. Additionally, we applied the iteration domain reconstruction of each process described in [19] to avoid transferring more data tokens than needed. We do not discuss this step in our solution approach.

Finally, we want to discuss how buffer sizes in FIFO channels of a PPN derived from a WDP program are determined. In our procedure we use the method of buffer sizes estimation presented in [3]. 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 WDP 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



```
for j = 1 to N,
  out_1 = f();
  write(o1, out_1);
endfor
```

```
for j = 1 to N,
  read(i1, in_1)
  for i = 1 to max_f,
    if i <= in_1,
      y_1[j,i] = F1()
      ctrl_c1[i] = j
      ctrl_c2[i] = i
    endif
   if j=N and i=5
      out_1 = ctrl_c1[i]
      out_2 = ctrl_c2[i]
      out_3 = y_1[ctrl_c1[i], ctrl_c2[i]]
      write(o1, out_1)
      write(o2, out_2)
      write(o3, out_3)
    endif
  endfor
endfor
```

```
if max_f >= 5,
  read(i1, in_1)
  read(i2, in_2)
  read(i3, in_3)
else
  in_1 = N+1
  in_2 = max_f+1
endif
if in_1 <= N & in_2 == 5,
   in_4 = in_3
 else
   in_4 = 0
 endif
[] = F2( in_4 )
```

Fig. 10: The final PPN derived from the program in Figure 5.

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 WDP programs data dependency relations are not static as some of the statements are guarded by dynamic *if*-conditions. 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 use the method from [3] to determine buffer sizes. To be able to handle the dynamism of WDP programs we have to follow a conservative strategy, i.e., we have to overestimate buffer sizes to provide enough space to run all possible instances of the dynamic program. In our procedure, we modify the iteration domains of input/output ports of all FIFOs, such that all dynamic *if*-conditions defining any of these iteration domains evaluate always to `true`. Then, we apply the procedure from [3] to the resulted PPN.

IV. Experiments

In this section, we apply our solution approach introduced in Section III on the motivating example shown in Figure 3.

According to Step 1 of the solution approach, we modify the initial program shown in Figure 3 in the following way: for loop nests with dynamic upper bounds at lines 3–4, 8–9 and 14–15, we substitute the upper bounds with constants equal to the maximum values variables `Height` and `Width` may take. In the LSOD program, the maximum values of `Height` and `Width` are the maximum dimensions of a picture frame provided by a video camera device. In addition, for each such loop nest, we introduce two arrays X_i and X_j that capture the values of the dynamic upper bounds at run-time. For example, the loop nest at lines 3–7 in the initial program in Figure 3 is modified into the loop nest in Figure 11.

By applying Steps 2 to 4 of our approach to the newly created program, we build the PPN depicted in Figure 12. The topology of this PPN consists of 5 processes, 3 *data channels* shown as solid lines that are used to exchange data between processes, 4 *data channels* shown as dotted lines used to propagate upper bounds `Height` and `Width`, and 3 *control channels* shown as dashed lines used to propagate control variables. The latter channels are defined by control variables as described in Step 3 of our solution approach.

Although our approach is general and will derive a PPN for any affine nested loop program with dynamic loop bounds, for some programs the derivation of more optimal PPN is possible using an

```
for k = 1 to Targets,
  [Height,Width] = getLSODTarget()
  X_j[k] = Height+1
  X_i[k] = Width+1
  for j = 0 to max_Height,
    for i = 0 to max_Width,
      if (j <= X_j[k] && i <= X_i[k])
        img[j,i] = readTarget()
      endif
    endfor
  endfor
...
endfor
```

Fig. 11: The LSOD application as a WDP.

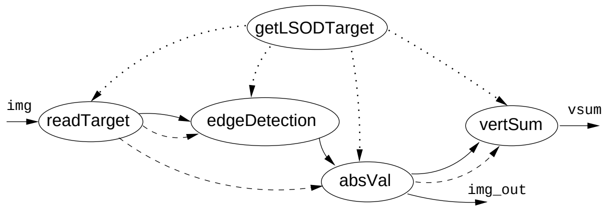


Fig. 12: The PPN derived from the LSOD program.

ad-hoc approach. Let us consider the LSOD application again. The LSOD application shown in Figure 3 is simple enough such that we can create the topology of the LSOD application's PPN just by inspecting its source code. This topology is comprised by the same nodes and data channels, as the PPN derived by our approach and depicted in Figure 12, *except* the control channels shown as dashed lines. This is possible due to the fact that all dynamic upper bounds of all loops in the program are the same. Therefore, once the values of variables `Height` and `Width` are determined at run-time at line 2 in Figure 3, the rest of the LSOD application turns into a SANLP program without loop-carried dependencies, and the corresponding PPN does not need any control channels.

We see, that our solution approach applied to the LSOD application derives a more complex PPN than the PPN obtained using the ad-hoc approach. This more complex PPN contains more control channels which leads to larger memory usage and increased communication control overhead at run-time. This is the price a system designer has to pay for using our general approach. However, in many cases the derivation of a PPN using an ad-hoc approach might be very difficult and time consuming. Moreover, the number of control channels in the ad-hoc derived PPN will be the same as in the PPN derived using our procedure. Therefore, although our approach may not give an optimal solution immediately, it provides very fast a valuable information to a system designer about the degree of parallelism present in the initial application specification. Additionally, our general approach may be considered as a reference point for further optimizations.

V. Related work

The work presented in this paper is an extension to previous works on systematic and automated derivation of process networks from affine nested loops programs. That is, Turjan et al. [20] proposed an automated derivation of process networks from *static* affine nested loop programs. In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Stefanov [16] further developed a procedure of process network derivation from more relaxed class of affine nested loop programs called *Weakly Dynamic Programs*. 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, our approach presented in this paper deals with affine nested loop programs with loop bounds determined at run-time.

Knobe and Sarkar [21] proposed a procedure for converting nested loop programs into a single assignment form that they called Array Static Single Assignment (ASSA). Their procedure accepts as an input a more general class of nested loop programs than the programs considered in this paper (`Dynloop` and WDP). Because of this and the fact that most of the analysis presented in [21] is done at run-time, their approach produces a large overhead in terms of memory usage and execution time.

The LooPo compiler [22] deals with parallelization of more general class of nested loop program than the class we consider in this paper. 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 [23].

In contrast to [21] and [22], 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.

VI. Conclusions

In this paper, 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 (PPN). This approach can be implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential programs. The approach presented in this paper includes only basic techniques that have to be applied in order to derive a PPN automatically from a `Dynloop` program. In some cases our approach may create more control channels than needed. 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. This fact indicates that some optimization techniques have to be added to our approach in order to improver the quality of the generated PPNs in terms of communication control overhead. We consider such optimization techniques as future work.

References

[1] G. Martin, "Overview of the MPSoC Design Challenge," in *Proc. DAC*, Jul. 2006.

[2] A. Mihal and K. Keutzer, "Mapping Concurrent Applications onto Architectural Platforms," in *Networks on Chips*, A. Jantsch and H. Tenhunen, Eds. Kluwer Academic Publishers, 2003, pp. 39–59.

[3] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: a tool for improved derivation of process networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 19–19, 2007.

[4] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, ser. LNCS, vol. 1132, 1996, pp. 79–103.

[5] "To appear in handbook of signal processing systems, download via: https://lirias.kuleuven.be/handle/123456789/235370."

[6] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[7] T. Stefanov et al., "System Design using Kahn Process Networks: The Compaan/Laura Approach," in *Proc. DATE*, Feb. 2004, pp. 340–345.

[8] E. de Kock, "Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study," in *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, Kyoto, Japan, Oct. 2-4 2002, pp. 68–73.

[9] K. Goossens et. al, "Guaranteeing the Quality Of Services in Networks On Chip," in *Networks on Chip*. Kluwer Publishers, 2003, pp. 61–82.

[10] B. Dwivedi et. al, "Automatic Synthesis of System on Chip Multiprocessor Architectures for Process networks," in *Proc. CODES+ISSS*, Sep. 2004.

[11] J. Castrillon et al., "Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms," in *Proc. of DATE'2010*.

[12] W. Haid et al., "Efficient execution of kahn process networks on multiprocessor systems using protothreads and windowed fifos," in *Proc. of ESTIMedia*. Grenoble, France: IEEE, 2009, pp. 35–44.

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

[14] P. Feautrier, "Dataflow Analysis of Scalar and Array References," *Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[15] J.-F. Collard, D. Barthou, and P. Feautrier, "Fuzzy array dataflow analysis," in *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. Santa Barbara, California: ACM Press, 1995, pp. 92–101.

[16] T. Stefanov, "Converting Weakly Dynamic Programs to Equivalent Process Network Specifications," 2004, phD thesis, Leiden University, The Netherlands, ISBN: 90-9018629-8.

[17] A. Turjan, B. Kienhuis, and E. Deprettere, "Realizations of the extended linearization model in the compaan tool chain," in *Proceedings of the 2nd Samos Workshop*, Samos, Greece, Aug. 2002.

[18] D. Nadezhkin and T. Stefanov, "Identifying Communication Models in Process Networks Derived from Weakly Dynamic Programs," in *Proc. SAMOS X*, July 2010, pp. 372–379.

[19] A. Turjan, "Compiling nested loop programs to process networks," 2007, PhD thesis, Leiden University, The Netherlands.

[20] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating Affine Nested-loop Programs to Process Networks," in *Proc. CASES'04*, Washington D.C., USA, Sep. 23-25 2004.

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

[22] M. Griebl and C. Lengauer, "The loop parallelizer loopo," in *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jlich*. Forschungszentrum, 1996, pp. 311–320.

[23] M. Geigl, M. Griebl, and C. Lengauer, "Termination detection in parallel loop nests with while loops," *Parallel Comput.*, vol. 25, no. 12, pp. 1489–1510, 1999.