Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

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.

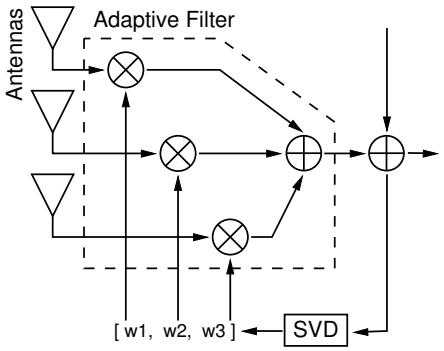
Recall, that in Section 1.1 we briefly introduced the main steps needed to translate a *static* sequential application into a PPN. Additionally, in Section 1.3 we showed that this approach cannot be used on dynamic applications. In this chapter we develop a new approach elaborating in more detail on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

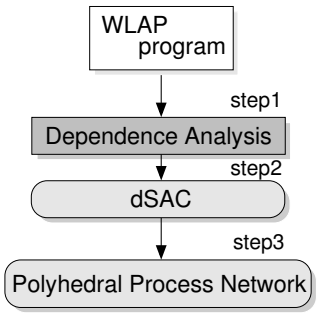
(b) An example of a WLAP program: the SVD algorithm

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

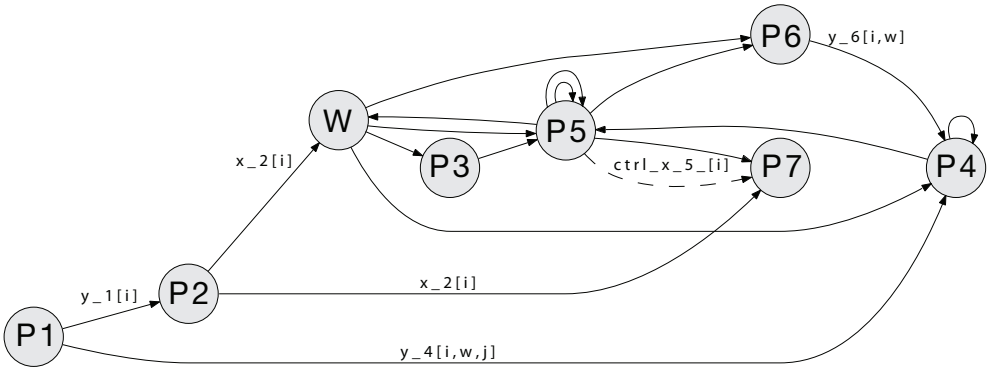


Figure 4.7: PPN representation of the program in Figure 4.6.

```
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.