Automatic Derivation of Polyhedral Process Networks from While-Loop Affine Programs

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

***Abstract*—The Process Networks (PNs) 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, signal processing, etc., that have adaptive and dynamic behavior which can not be expressed as SANLPs. Therefore, in order to handle more 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. 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 while-loops into input-output equivalent PPNs.**

I. Introduction

Whereas embedded multiprocessor hardware technologies have been making giant leaps in recent years, software technologies targeting these embedded hardware come always far behind. In particular, software developers are facing the problem of how to expose and utilize efficiently the parallelism available in applications to be able to fully utilize the power provided by multiprocessor embedded systems. The traditional approach where an application is specified using sequential model of computation (MoC) proved to be inefficient. The reason is that a sequential program does not match the way multiprocessor systems operate. A more promising approach is to specify an application using a parallel MoC. 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 [1] 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.

An example of a SANLP is given in Figure 2(a). 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 $F3()$ at line 6 which accepts i-th element of array `y[]` as an input argument. Providing just a pointer to array `y[]` in this case is not allowed. The above restrictions allow a compact mathematical representation of a SANLP using the well-known polyhedral model [2]. The SANLPs can be converted in an automated way into Polyhedral Process Networks (PPNs) [1].

The target PPNs is a special case of the Kahn Process Networks (KPNs) [3] model of computation. A PPN consists of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels using blocking read/write on an empty/full FIFO as synchronization mechanism. In addition, all code is expressed as parameterized polyhedrons [2], which 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 [4]. In comparison, computing buffer sizes is not possible for the more general KPN model. We are interested in the process network model because it provides a sound formalism, well suited for capturing and modeling of data-flow dominated applications in the realm of multimedia, imaging, and signal processing, that naturally contain tasks communicating via streams of data. Moreover, it has been already shown that process networks allow effective and efficient mappings of streaming applications to certain parallel execution platforms [5]–[10].

Many scientific, matrix computation, and signal processing applications can be specified as static affine nested loop programs (SANLPs), and therefore, the *pn* compiler [1] can be used to derive equivalent parallel PPN specifications. However, many multimedia applications, adaptive filters, iterative algorithms, etc. have adaptive and dynamic behavior which can not be expressed as SANLPs. In order to handle such dynamic applications, an important question should be addressed, namely, whether some of the restrictions of the SANLPs can be relaxed while keeping the

978-1-4577-2122-9/11/$26.00 ©2011 IEEE

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. We propose the following three relaxations to SANLP programs:

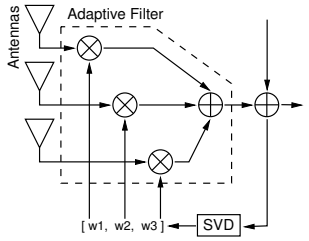
1) allow dynamic if-conditions;
2) allow for-loops with dynamic bounds;
3) allow while-loops.

In [11], [12], the first two relaxations have been considered, i.e., how to translate affine nested-loop programs with dynamic if-conditions and for-loops with dynamic bounds into input-output equivalent PPNs in an automated way. In this paper, we consider the third relaxation, while-loops, which is the most difficult of all these relaxations. The main contribution of this paper is a first approach for translation of *affine nested loop* programs with *while*-loops (WLAP) into input-output equivalent PPNs. This approach can be automated and implemented efficiently in a compiler that will help to reduce significantly the time for parallelizing sequential programs.

A. Motivating example

As a motivating example, we use a real-life application from the signal processing domain called Adaptive Beamforming (AB) [13]. With the description of the AB application below, we present a program that has the specific dynamic behavior we consider in this paper, 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 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

Fig. 1: Adaptive Beamforming and the SVD [14] 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 1(a). Pseudo-code of the SVD algorithm is illustrated in Figure 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 is an example of a program which has dynamic control. The program requires a while-loop at line 2 in Figure 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 [11] or for-loops with dynamic bounds considered in [12], the *pn* compiler [1] as well as techniques from [11], [12] are unable to handle the program in Figure 1(b). Therefore, 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 while-loops (WLAP) into input-output equivalent PPNs.

Handling the dynamic behavior of while-loops is more difficult compared to dynamic if-conditions and for-loops with dynamic bounds. A for-loop with dynamic loop bounds can be replaced by dynamic if-condition with some modifications as it has been shown in [12]. 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 paper, we demonstrate the analysis of while-loops in order to translate WLAPs into input-output equivalent PPNs.

The remaining part of the paper is organized as follows. In Section II we cover the related work. In Section III, we introduce some notations and present a technique used to analyze sequential programs with dynamic constructs. This is needed for better understanding of the solution approach we propose and discuss in Section IV. Finally, Section V concludes the paper.

II. Related work

The work presented in this paper is a significant extension to previous works [11], [12], [15] on systematic and automated derivation of process networks from affine nested loops programs. Turjan et al. [15] proposed an automated derivation of process networks from *static* affine nested loop programs (SANLPs). In SANLPs the memory array subscripts, loop bounds and conditional control structures are affine constructs of surrounding loop iterators, program parameters and constants. Stefanov [11] further developed a procedure for process network derivation from more relaxed class of affine nested loop programs called *Weakly Dynamic Programs* (WDPs). In this class of affine nested loops programs, the conditions in if-statements may be dependent on some data that is unknown at compile-time and may change at run-time. Nadezhkin et al. [12] further extended the class of WDP programs considered in [11] to programs with dynamic for-loop bounds (`Dynloop`) from which PPNs can be derived automatically. In contrast to the above mentioned techniques,

our approach presented in this paper deals with affine nested loop programs with *while*-loops that currently cannot be handled by [11], [12], [15] to derive PPNs.

There are a number of efforts which address the problem of while-loops parallelization. Raman et al. [16] devise the Parallel-Stage Decoupled Software Pipelining (PS-DSWP) multi-threading technique to extract pipeline parallelism from codes with irregular, pointer-based memory accesses and arbitrary control flow, which generally include while-loops. A parallel-stage allows to obtain pipeline parallelism from some stages executed in a DOALL fashion. In contrast, besides the pipeline- and iteration-level parallelism, our approach supports also task- and data-level parallelism. Moreover, we can generate parallel code for multi-processor systems with distributed memory.

Rauchwerger et al. [17] focused on parallelizing while-loops that are defined by one or more recurrences that can be detected at compile-time; a reminder that can be either analyzed statically or is unknown at compile-time; and one or more termination conditions. Although, they were able to parallelize a while-loop involving linked lists traversal, it is not shown how they would tackle more general while-loops, which we consider in our work.

A series of similar works on parallelization of while-loops is done by Griebl, Lengauer and Collard in [18]–[20]. Similar to our approach, they perform array dataflow analysis to expose data dependencies in an explicit way. Subsequently, they use space-time restructuring techniques to generate the code for speculative execution or software pipelining. Generally unscannable execution space that a while-loop provides, they scan with the help of run-time computable predicates, that are also used for detection of while-loops' termination. Besides introducing an overhead at run-time, these predicates limit the applicability of their approach to shared memory systems. In contrast, our parallelization approach targets multiprocessor systems with distributed memory.

Bijlsma [21] and Geuns [22] approach the problem of while-loops parallelization by considering an initial program with while-loops being in the local single assignment (LSA) form where all data dependencies are explicit. They implement the explicit data dependencies using circular buffers with overlapped read and write windows. Specifying a program in a LSA form can be very time consuming and error prone process because the system designer has to do the dependence analysis manually. We find this a very serious limitation of their work. By contrast, our approach has an automatic data-dependence analysis procedure which relieves the designer from the very difficult task to do the manual dependence analysis.

A different approach is taken by Benabderrahmane et al. [23] where they embed the control and exit predicates to the general data-dependent control-flow programs with while-loops. This predicates are used instead of while-loops as first-class citizens of the algebraic representation. Subsequently, a polyhedral representation is derived and code generation is performed from static program analysis. In this approach, hiding all while-loops in algebraic representations also diminishes the parallelism available in the initial program as less information is visible for analysis. By contrast, our technique exposes and utilizes all available parallelism.

```
0 parameter N 1 10

1  for i = 1 to N,
S1:  y[i] = F1()
3    for j = 1 to i,
S2:    y[i] = F2()
5    endfor
S3:  [] = F3( y[i] )
7  endfor
```

(a) Static Affine Nested Loop Program

```
0 parameter N 1 10

1  for i = 1 to N,
S1:  y[i] = F1()
3    while (...),
S2:    y[i] = F2()
5    endwhile
S3:  [] = F3( y[i] )
7  endfor
```

(b) Affine program with while-loop

Fig. 2: Examples of SANLP and WLAP programs.

III. Background

In this section, we introduce some notations used throughout the paper. Also, in Section III-B we formally describe a state-of-the-art technique called Fuzzy Array Dataflow Analysis (FADA) [24] used to analyze sequential programs with dynamic constraints. We introduce FADA because an important part of the solution approach in Section IV is based on this technique.

A. Notations

All arrays in a WLAP program are indexed by affine functions of static parameters and enclosing for-loop iterators only. An iteration vector $\vec{x}$ of a statement in a WLAP program is built from iterators of surrounding for- and while-loops. Although, an iterator for a *while*-loop may not be explicitly mentioned in the source code of a WLAP program, we can associate iterator $w : 0 \leq w$ with the *while*-loop. 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 2(b) is: $\mathbf{D}(S2) = \{(i, w) \mid 1 \leq i \leq N \wedge 1 \leq w\}$. An evaluation of a single statement W on iteration $\vec{x}$ is called an operation and denoted as $\langle W, \vec{x}\rangle$. By "$\prec$" we denote ordering of operations. An operation $\langle W, \vec{x}\rangle$ is evaluated before an operation $\langle R, \vec{y}\rangle$ ($\langle W, \vec{x}\rangle \prec \langle R, \vec{y}\rangle$) according to the program sequence if: 1) $\vec{x}$ lexicographically precedes $\vec{y}$; or 2) if $\vec{x} = \vec{y}$ and statement W precedes statement R in the program text. As described in [25], order "$\prec$" can be expanded to a system of linear inequalities. With "max" we denote the lexicographical maximum operator. The subvector of a vector $\vec{x}$ built from components k to l is written as: $\vec{x}[k..l]$.

B. Fuzzy Array Dataflow Analysis

In this section, we formally describe the FADA analysis. We introduce FADA because it is an important part of our solution we present in Section IV. FADA allows for the compile-time dependence analysis of programs where arbitrary *if*-conditions, for-loops and *while*-loops are allowed. 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 program depicted in Figure 2(b), the purpose of the dependence analysis is to find whether statement S_3 depends on statements S_1 or S_2 via array `y[]` and at which iterations. Or in other words, for every element of array `y[]` read at a given iteration of statement S_3, the dependence analysis finds which statement, S_1 or S_2, 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 statement W and R of a WLAP program. Operation $\langle W, \vec{x}\rangle$ writes to and operation $\langle R, \vec{y}\rangle$ reads from the same array. Moreover, statement W is enclosed in a *while*-loop at depth d. As a running example, consider Figure 2(b): statements S_2 and S_3 are W and R, respectively; statement S_2 is enclosed in the while-loop at depth 1. The iteration vector of statement S_2 is $\vec{x} = (i, w)$. To find whether operation $\langle W, \vec{x}\rangle$ is a source for operation $\langle R, \vec{y}\rangle$, we need to build and solve a system of linear inequalities:

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

First, we explain the meanings of constraints $(c2)$ and $(c3)$. Constraint (c2) specifies that if there is a dependency between two operations, both have to access the same array element. To access an array element, operation $\langle W, \vec{x}\rangle$ uses an affine indexing function $\mathcal{I}_W()$ and operation $\langle R, \vec{y}\rangle$ uses an affine indexing function $\mathcal{I}_R()$. The (c3) constraint determines an order of operations, i.e., source operation $\langle W, \vec{x}\rangle$ has to be evaluated *before* operation $\langle R, \vec{y}\rangle$.

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

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 $\langle R, \vec{y}\rangle$ from the same element occurs. Therefore, the source operation is the lexicographical maximum between all operations satisfying system $\mathbf{Q}_{WR}(\vec{y}, (\vec{\alpha}, \beta))$:

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

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 [24]:

$$\begin{aligned}\sigma(\langle R, \vec{y}\rangle,\ & (\vec{\alpha}, \beta)) = \\ & \max\{\langle W_k, \mathbf{K}_{W_k R}(\vec{y}, (\vec{\alpha}, \beta))\rangle \mid k \in [1, m]\}.\end{aligned} \tag{3}$$

For example, consider again the WLAP depicted in Figure 2(b). There are two statements S_1 and S_2 writing to array `y[]` and one statement S_3 which reads from it. For every pair S_1S_3 and S_2S_3 we build the systems of linear inequalities (1) which are depicted in Table I. To capture all evaluations of statement S_2, we introduce new iterator w which corresponds to the while-loop at line 3. For pair S_1S_3 all operations of statement S_1 are known and thus, a parameter is not introduced (see system $\mathbf{Q}_{S_1S_3}(i_3)$ in Table I). However, for pair S_2S_3 (see system $\mathbf{Q}_{S_2S_3}(i_3, (\alpha, \beta))$ in Table I), we introduce parameters α and β as shown in system (1), because statement S_2 is surrounded by the while-loop at line 3 in Figure 2(b) and, thus, exact operations of S_2 cannot be determined at compile-time. These parameters are used to designate at which iteration of S_2 a writing to the array `y[]` may occur. Values of the parameters are determined at run-time.

$\mathbf{Q}_{S_1S_3}(i_3)$	$\mathbf{Q}_{S_2S_3}(i_3, (\alpha, \beta))$	
$1 \le i_1 \le N$	$1 \le i_2 \le N \wedge$ $i_2 = \alpha \wedge 1 \le w \le \beta$	(c1)
$i_1 = i_3$	$i_2 = i_3$	(c2)
$\langle S_1, i_1\rangle \prec \langle S_3, i_3\rangle$	$\langle S_2, (i_2, w)\rangle \prec \langle S_3, i_3\rangle$	(c3)

TABLE I: Examples of system (1) for S_1S_3 and S_2S_3 pairs.

Approximated sources in S_1S_3 and S_2S_3 pairs are found by solving the parametric integer linear problems (PILPs) formulated in Table I. The "max" source defined in Equation 3 is determined by the recurrent algorithm of combining direct dependencies described in Section 5.2 of [24]. Thus, the source operation for statement S_3: $\sigma(\langle S_3, i_3\rangle), (\alpha, \beta))$ is:

$$\begin{aligned}\textbf{if}\quad i_3 = \alpha \wedge \beta \ge 1\quad & \textbf{then}\quad \langle S_2, (\alpha, \beta)\rangle \\ & \textbf{else}\quad \langle S_1, i_3\rangle.\end{aligned} \tag{4}$$

From Solution 4 above, we see that for any read operation $\langle S_3, i_3\rangle$ there are two data sources: statements S_1 or S_2. When for a given iteration i_3 of statement S_3, there is an iteration of statement S_2: $(i_2, w) = (\alpha, \beta)$, such that for $i_3 = \alpha$ there was at least one iteration of the while-loop, i.e., $\beta \ge 1$, then the source is statement S_2. Otherwise, the source is statement S_1. Solution 4 is approximated, because it depends on parameters (α, β) that are determined at run-time.

IV. Solution Approach

In this section we present our compile-time approach for translating WLAP programs into input-output equivalent PPNs. The approach consists of four steps. First, we find all data-dependency relations in the initial WLAP program by applying the FADA analysis on it. Recall that the result of the analysis is approximated, i.e., it depends on parameters which values are determined at run-time. Second, based on the results of the analysis, we transform the initial WLAP into a *dynamic Single Assignment Code* (dSAC) representation. dSAC was proposed in [11] as an extension of the SAC [25]. 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

$\mathbf{Q}_{S_2S_7}(i_7)$	$\mathbf{Q}_{S_3S_7}(i_7,\alpha,\beta)$	$\mathbf{Q}_{S_5S_7}(i_7,\alpha,\beta)$	
$1 \le i_2 \le N$	$1 \le i_3 \le N \wedge$	$1 \le i_5 \le N \wedge$	
	$i_3 = \alpha, 1 \le w_3 \le \beta$	$i_5 = \alpha, 1 \le w_5 \le \beta$	(c1)
		$i_5 + 1 \le j_5 \le N+1$	
—	—	—	(c2)
$\langle S_2,(i_2)\rangle \prec \langle S_7,(i_7)\rangle$	$\langle S_3,(i_3,w_3)\rangle \prec \langle S_7,(i_7)\rangle$	$\langle S_5,(i_5,w_5,j_5)\rangle \prec \langle S_7,(i_7)\rangle$	(c3)
	if $\beta \ge 1 \wedge 1 \le \alpha \le i_7$	**if** $\beta \ge 1 \wedge 1 \le \alpha \le i_7$	
$\langle S_2,(i_7)\rangle$	**then** $\langle S_3,(\alpha,\beta)\rangle$	**then** $\langle S_5,(\alpha,\beta,N+1)\rangle$	SOLUTIONS
	else $\bot$.	**else** $\bot$.	

TABLE II: Systems of linear inequalities (1) for pairs S_2S_7, S_3S_7 and S_5S_7 in the program in Figure 3.

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 [11], [12] for programs containing dynamic if and for-loops, whereas, in this paper, we present an extension to these procedures concerning while-loops. In the last fourth step, the topology of the corresponding PPN is derived, as well as the code executed in each process. All these steps can be represented as transformation of polyhedrons which we use for modeling the initial program and the target PPN. In the remaining part of this section, we describe the four steps in more detail and we also illustrate our solution approach with the example shown in Figure 3.

```
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
```

Fig. 3: A complex example of a WLAP program.

A. Step 1 (FADA analysis)

The formal description of the FADA algorithm has been given in Section III-B. In this step of our solution approach, we demonstrate the application of the FADA analysis on our running example in Figure 3.

Consider the WLAP program in Figure 3. 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 S_7. 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 3, 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 S_7 are in statements S_2, S_3 and S_5. Therefore, in order to find the source operation for statement S_7 we need to apply the FADA algorithm presented in Section III-B on pairs S_2S_7, S_3S_7 and S_5S_7. According to FADA, for all these pairs we build the systems of linear inequalities shown in Table II which correspond to Equation 1. Constraint $c1$ in Table II describes all possible source iterations of statements S_2, S_3 and S_5. 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 S_5 and iteration point (i_3, w_3) of statement S_3 when writing to scalar x may occur.

Solutions to the three parametric integer linear problems stated in Table II are shown in the last row of Table II. For example, in pair S_5S_7 the source operation for x is statement S_5 if condition $\beta \ge 1 \wedge 1 \le \alpha \le i_7$ evaluates to true. Otherwise, the source for x is not statement S_5 which is designated by $\bot$. In this case, statement S_7 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 II, the approximated source operation defined in Equation 3 for scalar x read in statement S_7 is:

$$\sigma_{\mathrm{x}}(\langle S_7,(i_7,\alpha,\beta)\rangle) = \left| \begin{array}{l} \textbf{if} \quad (\beta \ge 1 \wedge 1 \le \alpha \le i_7) \\ \textbf{then} \quad \langle S_5,(\alpha,\beta,N+1)\rangle \\ \textbf{else} \quad \langle S_2, i_7\rangle \end{array} \right. \tag{5}$$

From Solution 5 above, we see that for read operation $\langle S_7,(i_7,\alpha,\beta)\rangle$ there are two possible source operations. Depending on the values of the parameter vector (α, β), the source operation is either in statement S_2 or in statement S_5. The values of the parameter vector will be determined at run-time.

Similarly, we find the source operations for the other statements. Figure 4 shows the source σ functions only for statements S_4, S_5, S_6 and W that include non-trivial dependencies that exist in the program in Figure 3.

B. Step 2 (Initial dSAC)

The solutions provided by FADA are used to transform the initial WLAP program in order to expose the identified depen-

```
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

```
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
```

(c) Final dSAC

Fig. 5: Examples of the initial dSAC, the modified dSAC with control variables and the final dSAC.

$$\sigma_{\mathrm{y}}(\langle S_4,(i_4,w_4,j_4)\rangle) = \left|\begin{array}{l}\textbf{if}\ \ (j_4 = i_4+1)\\ \textbf{then}\left|\begin{array}{l}\textbf{if}\ \ (w_4=1)\\ \textbf{then}\ \ \langle S_1,i_4\rangle\\ \textbf{else}\ \ \langle S_6,(i_4,w_4-1)\rangle\end{array}\right.\\ \textbf{else}\ \ \langle S_4,(i_4,w_4,j_4-1)\rangle\end{array}\right. \quad (6)$$

$$\sigma_{\mathrm{x}}(\langle S_5,(i_5,w_5,j_5)\rangle) = \left|\begin{array}{l}\textbf{if}\ \ (j_5 = i_5+1)\\ \textbf{then}\left|\begin{array}{l}\textbf{if}\ \ (w_5=1)\\ \textbf{then}\ \ \langle S_3,(i_5,w_5)\rangle\\ \textbf{else}\ \ \langle S_5,(i_5,w_5-1,N+1)\rangle\end{array}\right.\\ \textbf{else}\ \ \langle S_5,(i_5,w_5,j_5-1)\rangle\end{array}\right. \quad (7)$$

$$\sigma_{\mathrm{x}}(\langle S_6,(i_6,w_6)\rangle) = \langle S_5,(i_6,w_6,N+1)\rangle \quad (8)$$

$$\sigma_{\mathrm{x}}(\langle W,(i_W,w_W)\rangle) = \left|\begin{array}{l}\textbf{if}\ \ (w_W == 1)\\ \textbf{then}\ \ \langle S_2,i_W\rangle\\ \textbf{else}\ \ \langle S_5,(i_W,w_W-1,N+1)\rangle\end{array}\right. \quad (9)$$

Fig. 4: Source operations for statements S_4,S_5,S_6 and W of the WLAP program in Figure 3.

dencies in an explicit way. The transformed program shown in Figure 5(a) is in dynamic Single Assignment Code (dSAC) form. The dSAC is an extension of the SAC introduced in [25]. 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 3 and generate the dSAC in Figure 5(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 implement the source σ functions depicted in Solution 5 and in Figure 4 found by FADA in the previous step of our solution approach. These source σ functions should be interpreted as code lines determined by Solution 5 and the solutions in Figure 4. For example, variable `in_5_x` at line 13 in Figure 5(a) is assigned by the source σ_x function defined by Solution 7 in Figure 4. This solution finds a source for scalar `x` read in statement $S5$ at line 9 in Figure 3. The whole line 13 in Figure 5(a) should be interpreted as the code in Figure 6. The code represents the σ_x function defined by Solution 7. Similarly, the other σ functions are represented in the code of dSAC.

Additionally, we transform the while-loop at line 5 in the initial program in Figure 3 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 5(a). Second, we replace line 5 in the initial program in Figure 3 with line 13 in Figure 5(a) implementing the same condition function. The source σ_x function defined by Solution 9 in Figure 4 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 5(a) by Solution 5. At this line, a source operation for scalar `x` read in RHS of statement S_7 is determined. Solution 5 is approximate, as the potential source statement S_5 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

```
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
```

Fig. 6: An interpretation of σ_x function for statement $S5$.

S_5 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.

C. 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 5(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 5(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 5(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 5 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 5(b), this program is input-output equivalent to the initial program in Figure 3.

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 5(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 5(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 Step 4, 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 5(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 5(c). This program is input-output equivalent to the program in Figure 5(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) [25]. 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 5(b) into the program in Figure 5(c) by inserting one-dimensional arrays `ctrl_x_5_[i]` at lines 22 and 23. The program in Figure 5(c) 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 5(c) is functionally equivalent to our running example shown in Figure 3. In the next step, we explain how to generate a process network from the program in Figure 5(c).

D. 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 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 5(c).

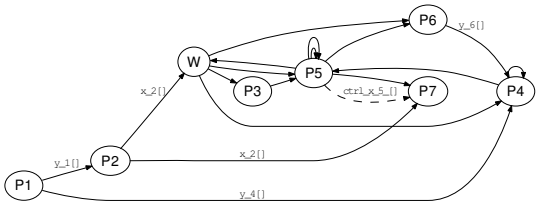


Fig. 7: PPN representation of the program in Figure 5(c).

```
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

Fig. 8: Internal source codes of processes W, S_5 and S_7.

Substep 1: Topology creation of a PPN

The PPN that corresponds to the program in Figure 5(c) is depicted in Figure 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 7 has 7 processes, $P1$–$P7$, that correspond to functions $F1$–$F7$ in Figure 5(c); and one process W which corresponds to the while-loop's condition function W at line 8 in Figure 5(c). The 18 communication channels correspond to data and control arrays in a single assignment form in the dSAC in Figure 5(c). Recall that data arrays in a single assignment are introduced after application of the FADA analysis on the WLAP program in Figure 3 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 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.

Substep 2: Code generation

Let us consider Figure 8, which illustrates the internal code structures of processes W, $P5$ and $P7$ of the PPN in Figure 7. Process W is an example of a process detecting the termination of the while-loop at line 5 in Figure 3. Process $P5$ 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.

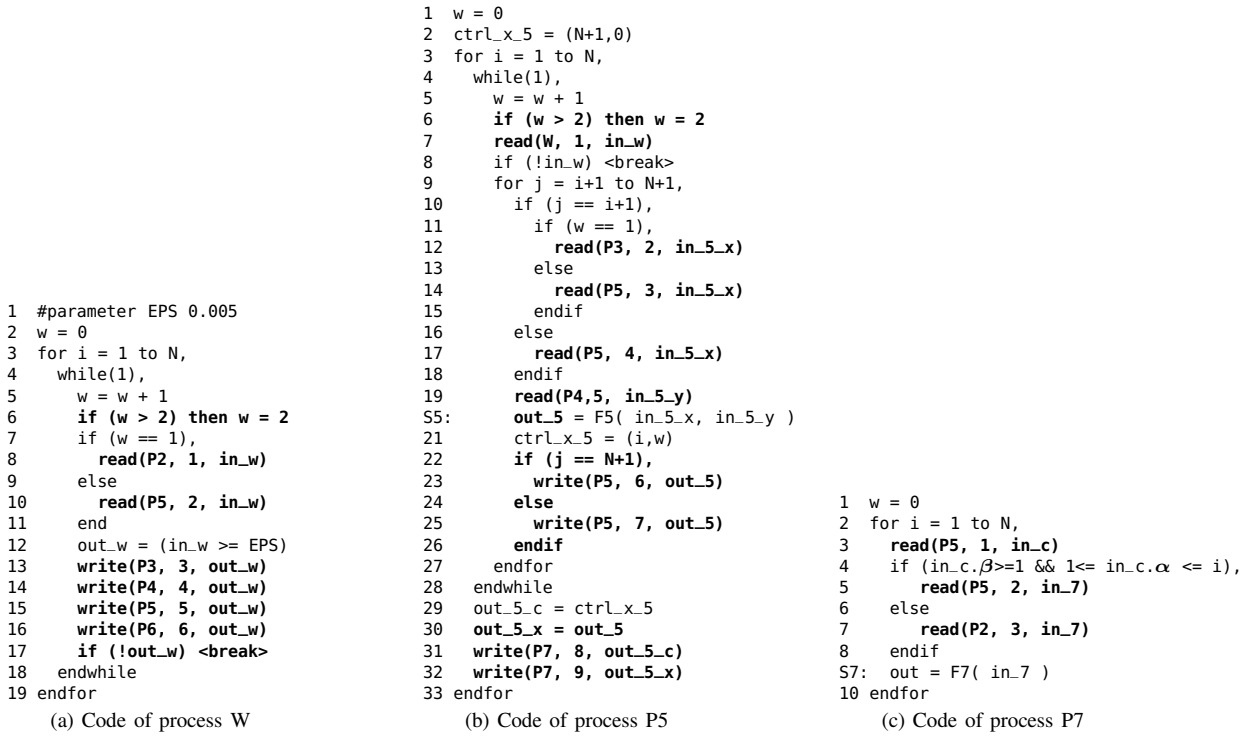
The internal code structure of each process is generated from the dSAC program derived in Step 3 of our solution approach. The code structure of each process is extracted from the code lines of the dSAC program. For example, all *non* highlighted (non-bolded) code lines in Figure 8 are taken from dSAC in Figure 5(c) expanding all σ source functions as explained in Section IV-B and illustrated in Figure 6. 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 8(a). Note, that lines 6–10 realize the interpretation of σ_x function defined in Solution 9 in Figure 4. 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 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*.

Substep 3: Linearization

Processes W, $P5$ and $P7$ depicted in Figure 8 are connected with communication channels which are the multi-dimensional arrays inherited from the dSAC shown in Figure 5(c). 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".



```
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

Fig. 9: Processes W, $P5$, and $P7$ after linearization of multi-dimensional arrays.

To implement the Linearization, we adapted the approaches proposed in [26], [27]. 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 8 is shown in Figure 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 P_5 in Figure 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 P_5, 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 derived from a WLAP program are determined. In our procedure we use the method of buffer sizes estimation presented in [1]. 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 [1] 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 [1] 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 [1] 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, and that w is not used in indexing we can use the method from [1] to determine the buffer sizes.

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 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 $\langle W, x\rangle$ and $\langle R, y\rangle$, 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 [24], the solution defined by Equation 1 in Section III-B 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 9(a) and 9(b).

V. Conclusion

In this paper, we presented an approach for automated translation of affine nested loops programs with while-loops (WLAPs) into input-output equivalent polyhedral process networks (PPNs). Leveraging the data dependence analysis, this approach extracts the maximum parallelism available in an application. Every step of our approach is the transformation of polyhedrons which we use for modeling. Therefore, 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. The approach presented in this paper includes only basic techniques that have to be applied in order to derive a PPN automatically from a WLAP program.

References

[1] 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.

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

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

[4] H. Nikolov, T. Stefanov, and E. F. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, 2008.

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

[6] 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.

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

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

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

[10] 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.

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

[12] D. Nadezhkin, H. Nikolov, and T. Stefanov, "Translating Affine Nested-Loop Programs with Dynamic Loop Bounds into Polyhedral Process Networks," in *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010*, Scottsdale, AZ, USA, October 2010, pp. 21–30.

[13] T.-J. Shan and T. Kailath, "Adaptive beamforming for coherent signals and interference," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 33, pp. 527–536, 1985.

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

[15] 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.

[16] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proc. 6th annual IEEE/ACM international symposium on Code generation and optimization*, New York, NY, USA, 2008, pp. 114–123.

[17] L. Rauchwerger and D. Padua, "Parallelizing while loops for multiprocessor systems," in *In Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[18] J.-F. Collard, "Automatic parallelization of while-loops using speculative execution," *Int. J. Parallel Program.*, vol. 23, pp. 191–219, April 1995.

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

[20] M. Griebl and C. Lengauer, "A communication scheme for the distributed execution of loop nests with while loops," *Int. J. Parallel Programming*, vol. 23, 1995.

[21] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit, "Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs," ser. SAMOS'09, 2009, pp. 140–148.

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

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

[24] D. B. Jean-Francois, J. francois Collard, and P. Feautrier, "Fuzzy array dataflow analysis," in *Journal of Parallel and Distributed Computing*, 1997, pp. 92–102.

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

[26] 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.

[27] 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.