Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

Chapter 3

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

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

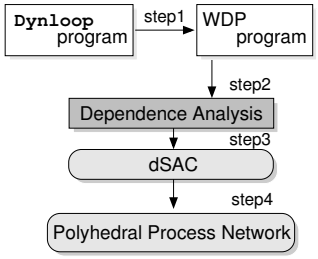
Recall, that in Section 1.1 we briefly introduced the main steps needed to translate a *static* sequential application into a PPN. Additionally, in Section 1.3 we showed that this approach cannot be used on dynamic applications. In this chapter we develop a new approach elaborating in more detail on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

```
1   parameter N 1 10;

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

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

(a) Dynloop program



(b) Parallelization approach

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

3.1 Solution Overview

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

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

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

3.2 Step 1 (Dynloop-to-WDP)

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

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

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

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

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

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

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

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

```
1  parameter N 1 10;

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

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

(a) Newly created WDP program

```
1  parameter N 1 10;

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

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

(b) Initial dSAC

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

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

3.3 Step 2 (FADA analysis)

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

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

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

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

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

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

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

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

3.3.1 Initial dSAC

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

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

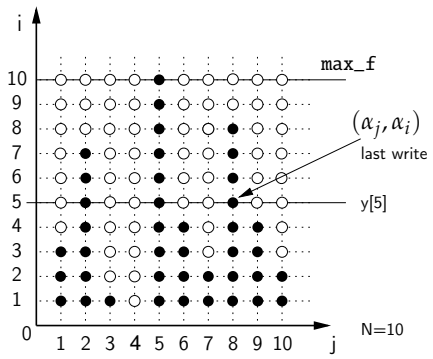


Figure 3.3: Representation of Solution 3.1 for the instance of the program in Figure 3.2(a).

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

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

3.4 Step 3 (Control arrays)

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

3.4.1 Local control arrays

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

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

```
1  parameter N 1 10;

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

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

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

```
1  parameter N 1 10;

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

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

(b) Modified dSAC code with new global control array

```
1  parameter N 1 10;

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

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

(c) Final dSAC

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

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

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

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

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

3.4.2 Global control arrays

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

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

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

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

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

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

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

3.5 Step 4 (PPN generation)

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

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

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

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

3.5.1 Topology creation of a PPN (substep 1)

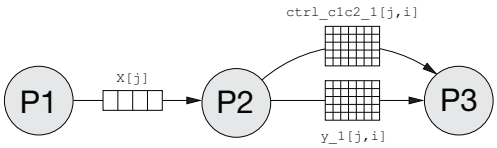


Figure 3.5: The topology of the PPN derived from the dSAC in Figure 3.4(c).

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

3.5.2 Internal code structure generation (substep 2)

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

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

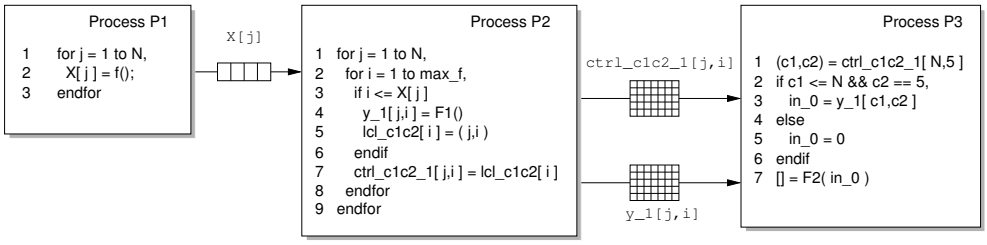


Figure 3.6: The internal code structure of each process in the PPN derived from the dSAC in Figure 3.4(c).

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

3.5.3 Linearization (substep 3)

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

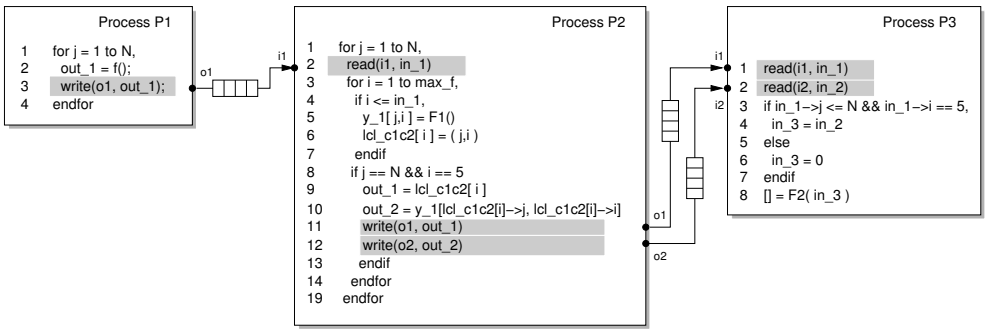


Figure 3.7: The final PPN derived from the program in Figure 3.1(a).

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

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

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

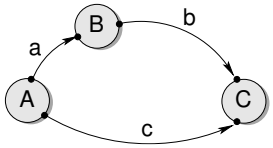
The approach presented in Chapter 5 identifies the communication characteristics of a data exchange in a pair of processes. Based on this information, the multi-dimensional array accesses are replaced with one-dimensional FIFO accesses. The result of the linearization applied on the multi-dimensional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

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

3.6 Calculation of deadlock-free buffer sizes

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

(a) Sequential Program



(b) PPN

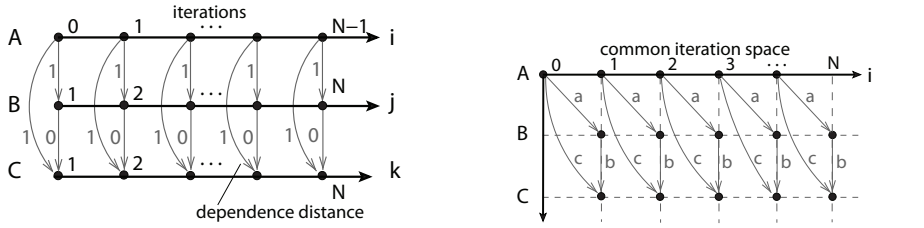
Figure 3.8: An example of a SANLP program and its PPN graph.

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

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

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

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



(a) Representation of data dependencies

(b) Schedule for minimum buffer sizes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Overhead Analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

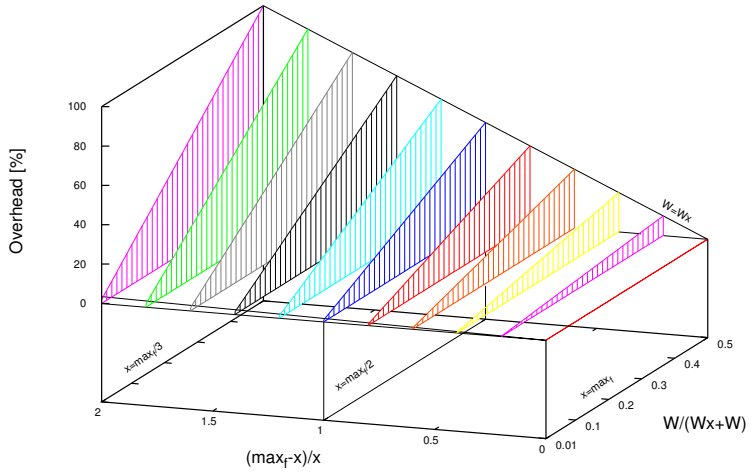


Figure 3.10: The amount of introduced overhead.

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

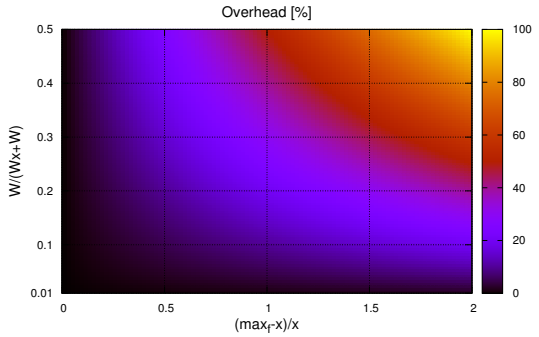


Figure 3.11: Overhead's color map.

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

3.8 Discussion and Summary

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

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

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

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

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

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

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