Parallelizing dynamic sequential programs using polyhedral process networks

Nadezhkin, D.

Chapter 1

Introduction

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

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

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

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

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

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

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

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

```
1 parameter N 10 100;

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

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

Figure 1.1: Pseudo code of a SANLP.

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

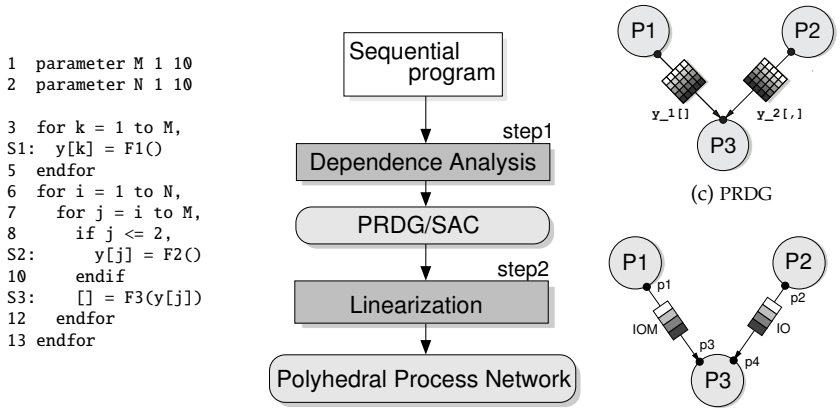
of *C*-language and emphasize that input programs considered in this dissertation are not language specific. A SANLP in Figure 1.1 consists of a set of statements and function calls, each possibly enclosed in loops and/or guarded by conditions. The loops do not have to be perfectly nested. All lower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

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

1.1 Automatic Derivation of Polyhedral Process Networks

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

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



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

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

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

1.1.1 Dependence Analysis

The goal of the *Dependence Analysis* step is to determine if evaluation of a statement depends on evaluation of other statements and to find these evaluations. For example, in the SANLP program depicted in Figure 1.2(a), the purpose of the dependence analysis is to find whether statement $S3$ depends on statements $S1$ or $S2$ via array y and at which particular iterations. Or in other words, for every element of array y read at a given iteration of statement $S3$, the dependence analysis finds which statement, $S1$ or $S2$, and at which iteration it writes data to the given array element. The most relevant algorithm to perform the analysis is the Exact Array Dataflow Anal-

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

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

$\mathbf{Q}_{\text{S1S3}}((i_3, j_3))$	$\mathbf{Q}_{\text{S2S3}}((i_3, j_3))$	
$1 \leq k \leq M$	$1 \leq i_2 \leq N \wedge i_2 \leq j_2 \leq M \wedge$ $j_2 \leq 2$	(c1)
$k = j_3$	$j_2 = j_3$	(c2)
`true`	$\langle S2, (i_2, j_2)\rangle \prec \langle S3, (i_3, j_3)\rangle$	(c3)

Table 1.1: Systems used in dependence analysis between statements S1S3 and S2S3 in the program shown in Figure 1.2(a).

Having these systems formulated, the data dependence algorithm finds the lexicographical maximum between all vectors satisfying the systems. Finding a lexicographical maximum means to solve the Parametric Integer Linear Problems (PILPs), where **objective** is the *lexmax* function **subjected to** conditions stated by systems $\mathbf{Q}()$. The solution to find the maximum point for a given convex domain defined by the systems is based on the dual simplex method [16] that is implemented in open-source libraries such as *isl* [17], Parma Polyhedral Library [18], and PIPLib [19]. Finally, the source operation for the data read by statement S3 is:

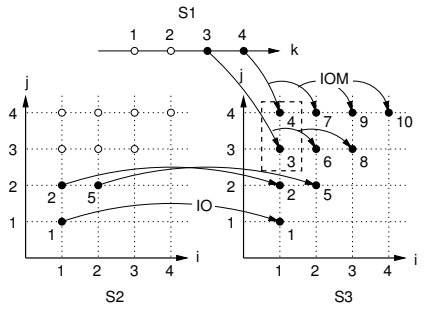
$$\begin{array}{lll} \textbf{if} \quad j_3 \leq 2 & \textbf{then} & \langle S2, (i_3, j_3)\rangle \\ & \textbf{else} & \langle S1, (j_3)\rangle. \end{array} \tag{1.1}$$

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

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

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

(a) SAC code

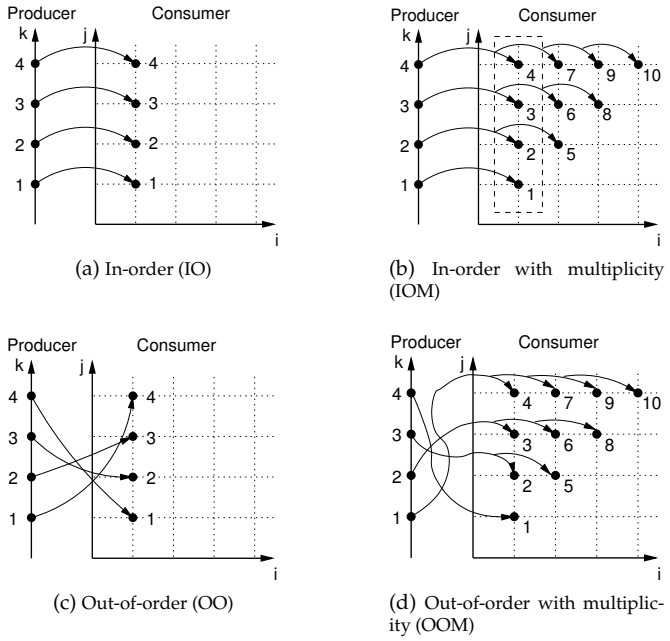


(b) Data dependencies

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

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

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



(a) In-order (IO)

(b) In-order with multiplicity (IOM)

(c) Out-of-order (OO)

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

Figure 1.4: Types of communication models in a PPN.

in Chapter 2.

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

1.1.2 Linearization

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

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

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

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

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

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

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

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

(a) Weakly Dynamic Program (WDP).

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

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

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

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

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

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

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

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

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

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

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

1.2 Problem statement

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

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

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

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

The first relaxation:

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

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

The second relaxation:

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

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

The third relaxation:

III. while-loops are allowed in a program.

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

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

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

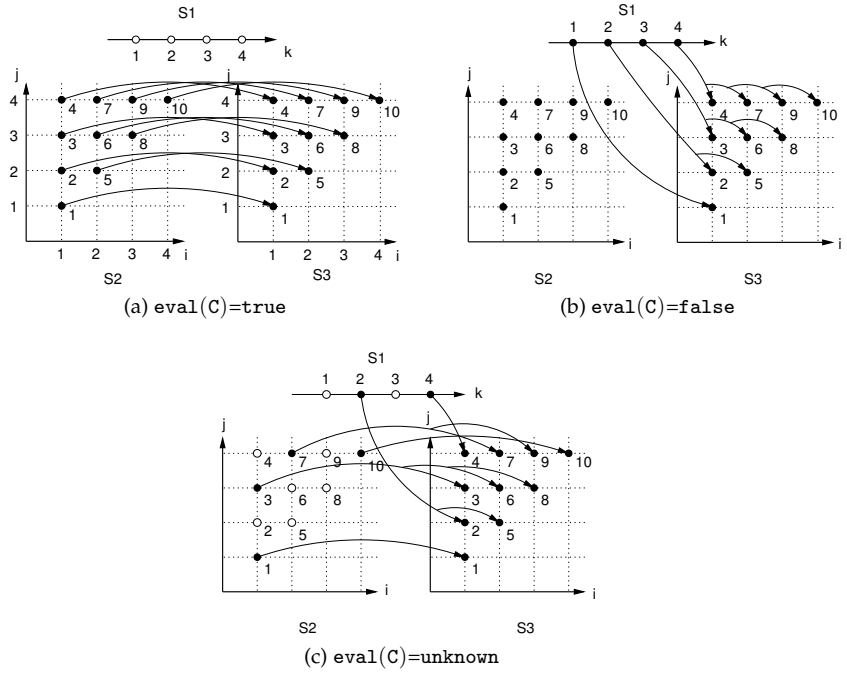


Figure 1.6: Dependency relations in a dynamic program.

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

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

1.3 Motivation and challenges

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.4 Research Contributions

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

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

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

1.5 Related Work

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.6 Dissertation Outline

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

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

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

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

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

port the results we obtained in this case study.

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