A compile time based approach for solving out-of-order communication in Kahn Process Networks

Alexandru Turjan Bart Kienhuis Ed Deprettere
Leiden Institute of Advanced Computer Science (LIACS),
Leiden, The Netherlands
e-mail: {aturjan,kienhuis,edd}@liacs.nl

Abstract

The Compaan *compiler framework automates the transformation of DSP applications written in Matlab into Kahn Process Networks (KPNs). These KPNs express the signal processing applications in a parallel distributed way making them more suitable for mapping onto parallel architectures. A simple instance of a generated KPN by Compaan is a Producer process that communicates with a Consumer process via a FIFO buffer, with the Consumer reading data from the FIFO using a blocking read. When the sequence of producing data is different from the sequence of consuming data, a simple FIFO is not sufficient to implement the communication. For such case, extra storage and control are needed at the consumer side. This paper presents a novel approach that determines at compile time whether a FIFO buffer is sufficient for every Producer/Consumer pair of a Compaan-generated KPN. For the case when the additional memory is required, we also provide an address generation mechanism at compile time. The presented approach is based on the Ehrhart theory.*

1 Introduction

The *Compaan* framework [8] automatically transforms digital signal processing applications, written in a subset of Matlab, into Kahn Process Networks. These KPNs express the signal processing applications in a parallel distributed way making them more suitable for mapping onto parallel architectures. These networks can be converted to VHDL and quickly synthesized to FPGAs [5] or mapped onto some parallel signal processing architectures [9] at a high level of abstraction to obtain first-order performance numbers.

The simplest instance of a Kahn Process Network is a *Producer* process that communicates with a *Consumer* process over an unbounded FIFO channel in which the Consumer reads data from the FIFO using a blocking read. A problem emerges if the order data is produced is different from the order data is consumed. Such situation leads to functional incorrect evaluation of the network and moreover it may lead to dead-lock. To avoid such situation, it is necessary to introduce a reordering mechanism that allows the Consumer to consume the data in the correct way. This paper presents a compile time approach to determine for a derived process network whether FIFOs are sufficient or additional reordering mechanisms are needed. For the case when a reordering mechanism is required, we also provide its model.

We assume that the process networks are derived using the Compaan toolset from nested loop programs written in Matlab. The toolset consists of three tools. The first tool transforms the initial

Matlab code into single assignment code (SAC), which represents the *dependence graph* (DG) of the initial nested loop program. The second tool converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data structure, which is a compact mathematical representation of the DG in terms of polyhedra. The third tool converts the PRDG into a process network by associating a process with each node of the PRDG. These parallel processes communicate with each other according to the data-dependency given in the DG.

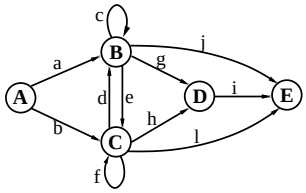

Figure 1. An example of a Polyhedral Reduced Dependence Graph.

An example of a PRDG graph is given in Figure 1. The nodes are { **A, B, C, ..., E** } and the edges { **a, b, c, ..., l** } between these nodes represent the dependencies. For example, node **E** has a data-dependency with **B**,**D** and **C**.

2 Producer-Consumer Pair

The communication between two processes, for example the communication between process **A** and **B** over the edge **a**, can be abstracted to an instance of the *Producer/Consumer pair* [1] shown at the top part of Figure 2. In this figure, the two processes contain nested-loop programs that communicate with each other using a global two-dimensional array *A[i,j]*. To obtain a Kahn Process Network, we have to replace this global array by a FIFO channel as shown in the lower part of Figure 2. This unbounded, one-dimensional FIFO channel is being accessed by means of a non-blocking *write* (implemented by `fifoPut` function) and a blocking *read* (implemented by `fifoGet` function).

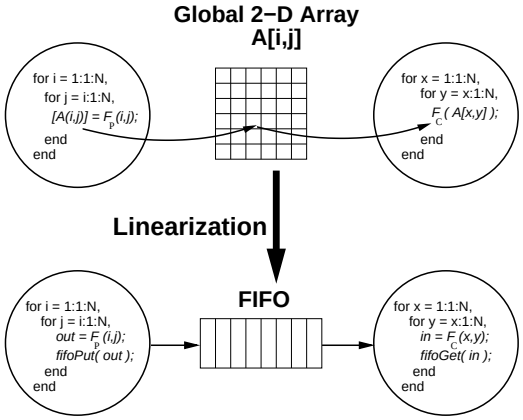

Figure 2. The Linearization Step and Linearization Model.

In general, the conversion from a PRDG, as shown for instance in Figure 1, to a process network entails that for each edge in the PRDG, the high dimensional data structure (e.g., A[i,j]) is

linearized into a single stream of data. We call this transformation *Linearization.* The model to do the linearization is given by the *linearization model.* The lower part of Figure 2 describes this model.

If we look at the nested loop programs for the Producer and Consumer in Figure 2, we see that the *for-statements* describe iteration spaces that have triangular shape. Although the *for-next* loops are parameterized in N, we show the iteration spaces for $N = 8$. The black dots inside the triangles represent *iteration points.* The triangular shapes can be described as a polytope and are called *domains.* In general, any system of for-statements can be expressed in terms of polytopes [6].

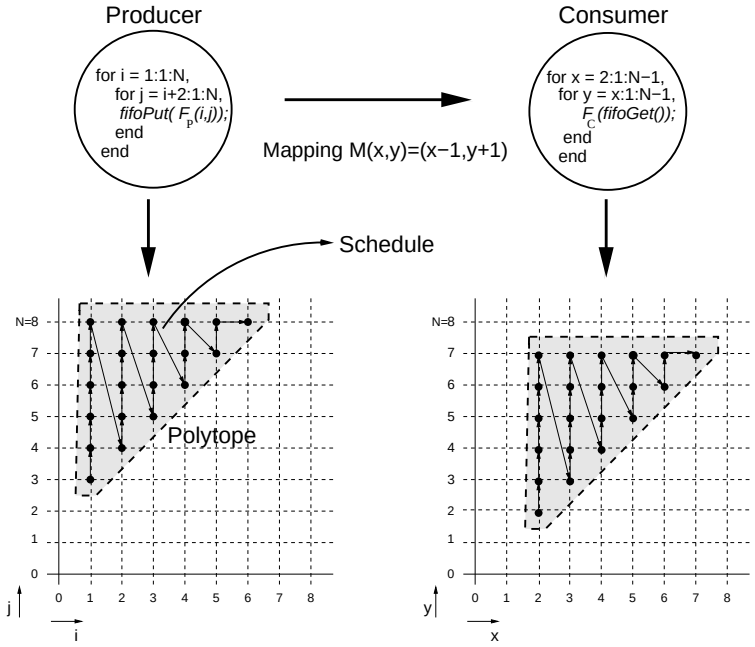

Figure 3. An Producer/Consumer pair.

The integer points inside a polytope are in principle not ordered. Nevertheless, the for-statements do impose a particular order as shown by the arrows in Figure 2. The order the *for-statements* step through the iteration space determines a particular *schedule.* There is an affine function between the Producer and the Consumer that represents the data dependency between the Consumer iteration points and the Producer iteration points. We call this a *mapping function.* In Figure 3, the mapping function is $M(x, y) = (x - 1, y + 1)$. At each iteration point of the Producer in Figure 3, a function F_p is evaluated and the result is sent over the FIFO to the Consumer. Similarly, at each iteration point of the Consumer, a value is read from the FIFO. This value is the argument for the function F_c, which is executed by the Consumer.

3 Different Schedules

The data that needs to be communicated between the Producer and the Consumer will be sent over a FIFO as part of the Linearization step. Because a FIFO is strictly ordered, the order data arrives at the Consumer side has to be the same order as it was produced by the Producer. According to the schedule of the Producer and the schedule of the Consumer, we distinguish two cases:

In order: the sequence of producing data is the same as the sequence of consuming data.

Out-of-order: the sequence of producing data is different from the sequence of consuming data.

3.1 In-order case

According to the schedules and mapping, the Producer/Consumer pair shown in Figure 3 describes an in-order case. The data being produced at the Producer side is read in the same order as given by the schedule at the Consumer side (See Figure 4). In such a case, a FIFO buffer is sufficient for the linearization of a Producer/Consumer pair.

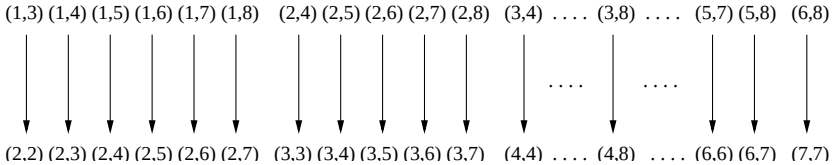

Figure 4. A sequence of producing and consuming data for the in-order case

3.2 Out-of-order case

However, if we interchange the indices *x* and *y* of the loops from the Consumer (see Figure 3) to force a different schedule, the nested loop program inside the Consumer becomes as follows:

```
for y = 2 : 1 : N-1,
  for x = 2 : 1 : y,
     Fc(fifoGet());
  end
end
```

Let's start to look at the first iteration from the Producer, which is `(1,3)` (See Figure 5). At this iteration, the Producer produces a token that is sent to the FIFO. This token is taken out from the FIFO at iteration `(2,2)` at the Consumer side. According to the mapping, the token produced at iteration `(1,3)` is consumed correctly by iteration `(2,2)`. The same sequence of events applies for the next iteration point from the Producer `(1,4)` and the Consumer `(2,3)`.

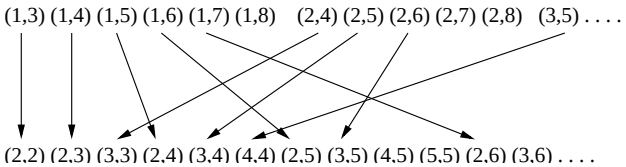

Figure 5. A sequence of producing and consuming data for the out-of-order case

A problem occurs when the Consumer wants to evaluate iteration point `(3,3)`. The token read from the FIFO is the token produced by the Producer at iteration `(1,5)`. However, according to the mapping function $M()$, the token produced at iteration `(2,4)` is required. The evaluation of the token related to iteration `(1,5)` by the Consumer would lead to an incorrect evaluation of the described process network. Hence, in case of an out-of-order schedule, the Linearization model proposed in Figure 3, is no longer valid. When the reordering problem appears, the Consumer should restore the desired order of the tokens.

Although we forced a different schedule to illustrate out-of-order execution, we remark here that the out-of-order problem is a consequence of the way the original Matlab is written. In the Matlab code the higher dimensional global array would also be accessed in an out-of-order fashion. Because Matlab is a sequential language, this reorder problem is not an issue, but if you go to processes running in parallel, this data reordering becomes very much a problem, as illustrated in Figure 5.

4 Solution of the reordering problem

In the conversion of a PRDG to a process network, we first need to be able to detect whether or not the linearization model is enough to realize the linearization step for an edge in a PRDG. If the regular linearization model cannot be used, i.e., a FIFO is not sufficient, as shown by the *out-of-order* example, we need a special reordering mechanism to realize the correct communication. We first discuss a compile time solution to detect for a given producer/consumer pair if we are in the in-order or the out-of-order case. Next, we discuss an extension of the linearization model for the out-of-order case.

4.1 Detection of out-of-order Producer/Consumer pairs

To detect if a FIFO is sufficient in the linearization step, we have to be able to compare the order data is produced with the order it is consumed. This requires that we establish a basis allowing us to reason about the order of the iteration points of a nested loop program. As such basis, we would like to find a function that associates to each iteration point a number that expresses the order this point it is executed relative to other iteration points. We call this function the *rank function.*

Suppose we are able to establish such a *rank* function for a Consumer process and for a Producer process. The relationship between the Producer domain and the Consumer domain is given by the mapping function $M()$. If we compose the rank function of the Producer with this mapping, we establish a function that for each Consumer iteration point gives the order in which the needed token arrives through the FIFO. We call this function the *read function*:

$$read(x_c) = rank_P(x_p) \circ F(x_c).$$

If the *rank* function is equal to the *read* function for all the iteration points at the Consumer side, then each two corresponding points (one from the Producer process x_p and one at the Consumer process x_c) have the same order in the execution of their domain loops:

$$read(x_c) - rank_C(x_c) = 0; \;\; \forall x_c \in Consumer.$$

Thus, if these two functions are equal, we conclude that a FIFO is enough; the Consumer can read directly from a FIFO and no additional reordering memory is necessary.

4.2 Extended Linearization Model

If the linearization step can not be used because of the out-of-order problem, we have to extend the communication mechanism between a Producer and Consumer. The extension is to include additional memory at the Consumer to restore the desired order of the tokens. This leads to a new linearization model, which we call the *Extended Linearization Model*. We can do this extension without violating the process network semantics, because this memory can be seen as the internal

state of a Process [7]. The organization of the Extended Linearization Model is shown in Figure 6 and consists of three modifications to the Consumer process:

A Change a *fifoGet* call in a nested loop program into a *control.getFrom(i,j)* call.

B Add some random accessible memory (RAM).

C Add a controller unit that takes care of restoring the order of tokens by providing an implementation of the $getFrom(x_c)$ function.

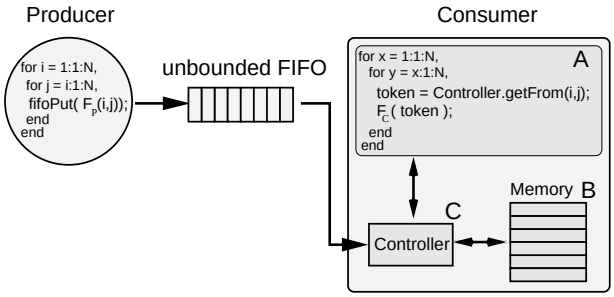

Figure 6. Extended Linearization Model

5 Deriving the Rank Function

To detect the schedule between a Producer and Consumer, we need to find the $rank$ function. In order to find this function, we use the Ehrhart theory that expresses the number of integral points inside of a polytope as a polynomial expression [4]. This theory has recently been extended to work also for parameterized polytopes [2], in which case we find pseudo-polynomials. Such pseudo-polynomials have non-constant coefficients.

We defined the rank of an iteration point x as the number of iteration points of the for-statements that were executed before iteration point x. The set of the points x' that have this property, are the points that are lexicographically smaller than point x (i.e., $x' \prec x$). Let a domain be a convex polyhedron P of dimension d, then using the lexicographical order, we can define the next set of polytopes:

$$J_l(x) = \{x^{'} \in Z^d \cap P : x_1 = x_1^{'} \wedge \cdots \wedge x_{l-1} = x_{l-1}^{'} \wedge x_l < x_l^{'}\}.$$

where $l = 1$ to d and the *rank* of *x* becomes:

$$rank(x) = \sum_{l=1}^{d} |J_l(x)| + 1.$$

This function has the next properties:

- $rank : Z^d \rightarrow Z^+$ is uniquely determined by a subset of P: $P^{'} \subset P$.
- $rank : P \rightarrow Z^+$ is a bijective function which maps the domain P onto Z^+.

To calculate the number of integral points inside $|J_l(x)|$, we use the extended Ehrhart theory to obtain the pseudo-polynomial expression $P_l(x)$. If we sum all pseudo-polynomials from 1 to d, we

obtain the rank function as:

$$rank(x) = P_1(x) + P_2(x) + \ldots + P_d(x).$$

Suppose that we have derived $rank_P$ and $rank_C$ for a Producer/Consumer pair. Because we have the mapping $M()$ between the Producer and Consumer, we can derive the $read$ function. Because for a given domain P the rank function is uniquely defined by subset P' [4], it results that the only possibility such that $read = rank_C$ for all the Consumer iteration points, is that these functions should be identical:

$$read(x) == rank_C(x).$$

As a consequence, to decide if the $rank_C$ and the $read$ functions are equal for all Consumer iteration points, it is not necessary to compute and to compare their values for all the iteration points. Instead, a compile time comparison of the coefficients of the pseudo-polynomials that represents these functions is sufficient.

5.1 Example of the Rank function

Let us have a look at the following code as given below:

```
for i = 2 : 1 : N-1,
   for j = i : 1 : N-1,
       ...
   end
end
```

This nested for-loop program can be represented as a parameterized polyhedron P with a particular schedule. The polyhedron is determined by the boundaries of the loops. In this example the polyhedron P is represented by a surface in a two dimensional space as shown in Figure 7.

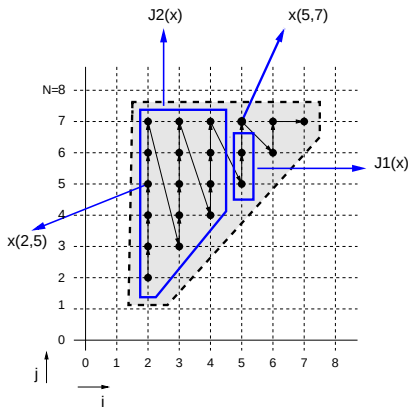

Figure 7. Ranking into a nested loop domain

The order of the iteration points is determined by the iterators of the for-loops; the i axis is the outer loop and the j axis is the inner loop. Because the points from the polyhedron P are the iteration points of the nested loop program, we can find the rank of $x(i, j)$ as the sum of the number of the integer points which are in the parameterized polyhedra $J_1(x)$ and $J_2(x)$. Because the derived polyhedra $J_l(x)$ and $J_2(x)$ are convex, we can use Ehrhart's theory to count how many

integral points are contained by the polytopes. Therefore, the rank function is the summation of P_1 and P_2 and is equal to

$$rank(i,j) \;=\; (i-2)*N \;\;+\;\; (-1/2*i^2 - 1/2*i + 2 + j).$$

This formula shows that the order of iteration point $x(5,7)$ has $rank(x(5,7)) = 17$. This rank number has been obtained by adding the number of the points in polytope $|J2(x)| = 15$ with the number of points in polytope $|J1(x)| = 2$. Hence, 17 iterations have been executed before iteration $x(5,7)$ is executed. The rank of point $x(2,5)$ is 4. Because $4 < 17$, we conclude that point $x(2,5)$ has been executed before point $x(5,7)$.

6 The Rank/Read functions at work

Let's apply the procedure discussed in Section 5 on our Producer/Consumer pair shown in Figure 3. First, we present an *in-order* case followed by an *out-of order* case.

6.1 The in-order case:

For the *in-order* case we obtain the *rank* in the Producer domain:

$$rank_P(i,j) = (i-1)*N + (-1/2*i^2 - 3/2*i + j). \tag{1}$$

Similarly the *rank* function at the Consumer side is:

$$rank_C(x,y) = (x-2)*N + (-1/2x^2 - 1/2*x + y + 2).$$

The mapping between the Consumer and the Producer is given by the affine transformation:

$$F((x,y)) = (x-1, y+1). \tag{2}$$

To obtain the $read_C$ function, we compose the $rank_P$ from the Producer with the mapping between the Consumer and Producer:

$$\begin{aligned} read &= rank_P(i,j) \circ F(x,y) \\ &= rank_P(x-1, y+1) \\ &= (x-2)*N + (-1/2*x^2 - 1/2*x + y + 2). \end{aligned}$$

We now subtract the rank function from the read function to see if they are identical:

$$read(x,y) - rank_C(x,y) = 0.$$

Since the result is zero, the $read(x,y)$ function and the $rank_C(x,y)$ are identical for all the points $(x,y) \in Z^2$. In conclusion, the order between Producer and Consumer is the same and only a FIFO buffer between the two processes is needed in the linearization step shown in Figure 3.

6.2 The out-of-order case:

Applying the same procedure for the *out-of-order* case. The code of the Consumer is the code given in Section 3.2. The $rank_C$ function at the Consumer side is:

$$rank_C(x, y) = 1/2 * y^2 - 3/2 * y + x.$$

Because we have not changed the domain of the Producer, it results that $rank_P$ is the same as for the *in-order* case (see Equation 1). The mapping between the Consumer and the Producer is still given by the affine transformation as given in Equation 2. Composing the function rank from the Producer with the mapping, the read function is:

$$\begin{aligned} read(x, y) &= rank_P(i, j) \circ F(x, y) \\ &= (x - 2) * N + (-1/2 * x^2 - 1/2 * x + y + 2). \end{aligned}$$

If we subtract the rank function from the read function, we see that they are not identical:

$$\begin{aligned} read(x, y) - rank_C(x, y) = \\ (x - 2) * N + (-1/2 * x^2 - 3/2 * x) + (-1/2 * y^2 + 5/2 * y) + 2. \end{aligned}$$

For example, if we look at iteration point $(3, 3)$ at the Consumer, the get $read(3, 3) - rank_C(3, 3) = 4$. Since this result is not equal to zero, we conclude that a FIFO buffer is not enough in the linearization step and we have to use the extended linearization model.

7 Realizing the extended Linearization model.

If we want to use the extended linearization model, the problem is to determine the behavior of the controller to restore the order of tokens. In order to restore the order, the controller has to determine at each consumer iteration point from where it has to read data: from the FIFO or from the reordering memory. For the case when it has to read from the FIFO, the consumer has to know how many tokens has to load from the FIFO into the reordering memory before the required token appears at the bottom of the FIFO and can be accessed. For the case when the consumer has to read from the memory, the consumer has to know the address where the needed data is located.

The consumer decides whether at iteration point x it has to read from FIFO and if so, how many tokens according to the next function:

$$\Delta(x) = read(x) - max(read(x_i)), x_i \prec x. \tag{3}$$

Depending on the value of $\Delta(x)$, the Controller acts as follow to obtain the needed token for iteration x:

If $(\Delta > 1)$, the controller moves Δ tokens from the FIFO into the reordering memory returning the last one to the consumer process, which is the needed token.

If $(\Delta == 1)$, the needed token is consumed directly from the FIFO and loaded into the reordering memory, because it may be needed by future iteration points.

If $(\Delta < 0)$, the needed token is consumed directly from the reordering memory from the address given by *read(x)*.

$x(i,j)$	$read(x)$	$max(read)$	$\Delta(x)$
(2,2)	1	0	1
(2,3)	2	1	1
(3,3)	7	2	5
(2,4)	3	7	-4
(3,4)	8	7	1
(4,4)	12	8	4
(2,5)	4	12	-8
(3,5)	9	12	-3

Table 1. The table determines what the Controller unit should do at an given iteration point *x*.

In Table 1, we show the values of the $read(x)$, $max(read(x))$, and their difference $\Delta(x)$ for a number of Consumer iteration points from the out-of-order example given in Figure 5. For the iterations $x(2,2)$ and $x(2,3)$, the consumer reads directly from the FIFO. For the iterations $x(2,4)$ and $x(2,5)$, the consumer accesses directly the reordering memory to get the desired token at the address given by the $read$ function. For the iterations $x(3,3)$, $x(3,4)$, and $x(4,4)$, the consumer has to read 5, 1, and 4 tokens respectively from the FIFO and store them in the local memory until the desired token arrives.

7.1 Controller Implementation

The main part of the Controller that delivers data to the Consumer is implemented by the `getFrom` function. The pseudo-code representation of this function is given below. In this code, the `getFrom` function consists of two parts. The first part determines the unique address to access the reordering memory using the function `read(x,y)` as defined in Section 4.1. The second part checks whether the reorder locations already contain the required token, i.e., whether the required token was previously read from the FIFO and stored in the reordering memory. Function Δ performs this check and is given by Equation 3. For a given iteration point (x,y), it returns the number of tokens that have to be read from the FIFO and written into the reorder memory in consecutive order (see also Table 1).

```
Token token getFrom(x,y) {
 address = read(x,y);
 d = Delta(x,y);
   if( d < 0 ){
       // token was already loaded into the memory
       return Mem.getAt(address);
   } else if ( d >= 1 ) {
       // return the d position from the FIFO
       return loadFromFifoIntoMem(d);
   }
}
```

The function Δ can be either implemented as a pseudo-polynomial expression that is evaluated for each iteration or precomputed and realized as a simple look-up table. However, the latter case can only be applied if the original application is no longer parameterized.

8 Implementation and Limitations

In Compaan, we implemented the procedure that generates the *rank* and *read* function for arbitrary domains. This implementation is based on the *enumerate* procedure from the Polylib library [3]. Using this procedure, we found the polynomial expressions of read and rank used in this paper. For many cases, derived from real DSP applications, we are able to find the proper pseudo-polynomial expressions. Nevertheless, the Ehrhart function implemented in Polylib is still under development and using the Compaan framework, we have found cases for which the Ehrhart implementation cannot find a solution and fails with "Degenerated Domain" errors. We have identified and isolated a number of issues and provided feedback on these issues to the designers of the Ehrhart function in Polylib.

In practice, the pseudo-polynomials obtained for the read and rank functions can be quite complex. To determine whether two pseudo-polynomials are identical is quite involved, but always possible.

The approach to determine the rank and read function is available when all the points contained inside of the polyhedron P (see Section 5) are indeed iteration points of a nested loop program. However, there are cases in which that does not occur. These situations occur when the for-statements in a Matlab program use a stride bigger than one, or some affine linear expressions in the Matlab program contains non-linear operations like *mod*, *div*, *floor* or *ceil*. These can lead to so-called holes. In that case, the rank function evaluates wrongly the order of the iteration points, because it doesn't handle correctly the integer points that do not belong to the original nested-loop program (holes), but that are part of the polyhedron [10].

9 Conclusions

Within the Compaan toolset, an important step is the conversion of a PRDG to a KPN. In this conversion, all nodes of the PRDG are replaced by parallel running processes and a FIFO buffer replaces the edges between the nodes. This is called the linearization step. We show that in general, we can abstract the communication between two processes to a simple Producer/Consumer pair. Normally, a FIFO buffer is enough to realize the linearization. However, there are cases that additional memory is required, namely when the order of producing data is different from the order of consuming data. This leads to an incorrect evaluation of the KPN and the network might even deadlock.

We presented in this paper a novel compile time technique for detecting whether a FIFO or additional reordering mechanism is required in the linearization step. Moreover, we provided an implementation of the additional reordering mechanism, called the extended linearization model. The strength of the presented approach is that for this extended linearization model, we can again derive an implementation at compile time.

The novelty in solving the linearization problem is the ability to expresses the order of iteration points in terms of a polynomial expression, which we call the rank function. Such a polynomial is obtained using Ehrhart theory. By comparing the coefficients of the derived read and rank functions, we can determine whether additional memory is needed in a Producer/Consumer pair and thus that a FIFO satisfies. If, however, these polynomials are not identical, additional reordering memory is required at the Consumer side.

We have implemented the rank function in software and have tested the presented approach on a set of real digital signal processing algorithms showing that the proposed approach is feasible.

References

[1] M Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall, 1982.

[2] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.

[3] Philippe Clauss and Vincent Loechner. Polylib. http://icps.u-strabg.fr/Polylib, February 2002.

[4] Eugène Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*. Birkhäuser Verlag, Basel, international series of numerical mathematics vol. 35 edition, 1977.

[5] Tim Harriss, Richard Walke, Bart Kienhuis, and Ed. F. Depettere. Compilation from matlab to process networks realized in fpga. In *Proceedings of the 35th Asilomar conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, November 4 – 7 2001.

[6] Peter Held. Functional Design of Data-Flow Networks, 1996. PhD thesis, Delft University of Technology, The Netherlands.

[7] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[8] Bart Kienhuis, Edwin Rypkema, and Ed Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.

[9] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System level design with spade: an m-jpeg case study. In *proc. Int. Conference on Computer Aided Design (ICCAD'01)*, pages 31 – 38, San Jose CA, USA, November 4 – 8 2001.

[10] Jurgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14:297–332, February 1993.