

DERIVING PROCESS NETWORKS FROM NESTED LOOP ALGORITHMS

EDWIN RIJKEMA and ED F. DEPRETTERE

*Leiden University, Niels Bohrweg 1
2333 CA Leiden, the Netherlands
{rijpkema,edd}@liacs.nl*

BART KIENHUIS

*University of California,
Berkeley, CA 94720, USA
kienhuis@eecs.berkeley.edu*

Received (received date)

Revised (revised date)

Communicated by (Name of Editor)

ABSTRACT

High level modeling and (quantitative) performance analysis of signal processing systems requires high level models for the applications (algorithms) and the implementations (architectures), a mapping of the former into the latter, and a simulator for fast execution of the whole. Signal processing algorithms are very often nested-loop algorithms with a high degree of inherent parallelism. This paper presents - for such applications - a suitable application model and a method to convert a given imperative executable specification to a specification in terms of this application model. The methods and tools are illustrated by means of an example.

1. Introduction

A new kind of embedded architectures is emerging that is composed of a microprocessor, some memory, and a number of dedicated coprocessors that are linked together via some kind of programmable interconnect (See Figure 1). These architectures are devised to be used in real-time, high-performance signal processing applications. Examples of these new architectures are the *Prophid* architecture [1], The *Jacobium* architecture [2], and the *Pleiades* architecture [3], to be used in respectively, video consumer appliances, adaptive array signal processing, and wireless mobile communications. These architectures have in common that they exploit parallelism using instruction level parallelism offered by the microprocessor and coarse-grained parallelism offered by the coprocessors. Given a set of applications, the hardware/software codesign problem is to determine what needs to execute on the microprocessor and what on the coprocessors and, furthermore, what should each coprocessor contain, while being programmable enough to support the set of applications.

The applications that need to execute on the architecture are typically specified using an imperative model of computation, most commonly C or Matlab. In Figure 1, for example, we show an algorithm written in Matlab. Although the imperative model of computation is well suited to specify applications, it does not reveal parallelism due to its inherent sequential nature. Compilers exist that are able to extract instruction level parallelism from the original specification at a very fine level of granularity. They are, however, unable to exploit coarse-grained parallelism offered by the coprocessors of the architecture. This makes the mapping of the applications onto the architecture difficult.

Instead, a better specification format would be to use an inherently parallel model of computation like *Process Networks* (PN) [4,5]. A PN models applications as a network

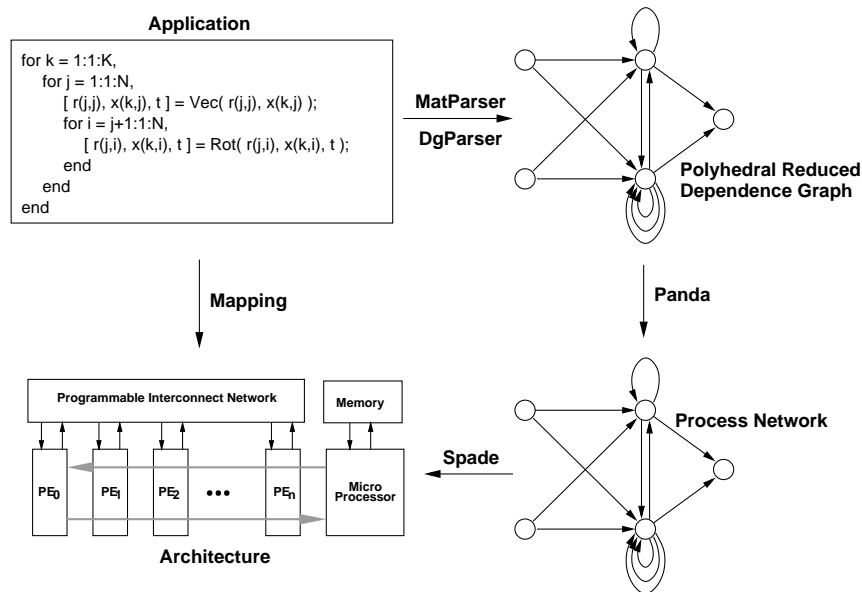


Fig. 1. Mapping the application onto an architecture is difficult because the Model of Computation of the application does not match the way the architecture operates.

of concurrently executing sequential processes. It expresses parallelism naturally from the very fine-grained to the very coarse-grained, and does not pre-impose any particular schedule. The mapping then becomes the assignment of the processes to the microprocessor and the coprocessors as shown by tools like ORAS [6], or SPADE [7]. Using these tools, a Y-chart [8] can be constructed, allowing the quality assessment of mappings on architectures.

This paper describes the *Compaan* tool set that automatically transforms certain Matlab applications into a process network description, as shown in Figure 1. It converts a Matlab application into a *polyhedral reduced dependence graph* (PRDG), that is subsequently converted into a *Kahn process network* (KPN) description [4]. The *Compaan* tool set is confined to operate on affine nested loop programs (NLPs) [9] that appear often in applications of interest.

The outline of the paper is as follows. Section 2 gives the *Y-chart* exploration scheme underlying our concept of modeling and simulation. Section 3 presents the Matlab-to-Process Network compiler *Compaan*. Section 4 briefly reviews the tools *MatParser* and *DgParser* that generate from a given imperative specification a single assignment code and a reduced dependence graph, respectively. Section 5 introduces the tool that generates the processes in the Process Network specification of an application.

2. The Modeling Concept

In line with Kienhuis et al. [10] we advocate that the development of heterogeneous architectures should follow a general scheme, which can be visualized by the Y-shaped figure in Figure 2(a): The *Y-chart*. In the upper right part of Figure 2(a) is the set of applications

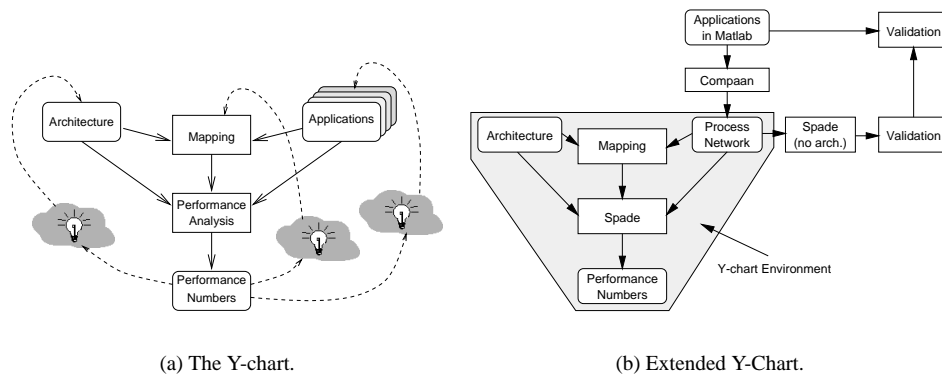


Fig. 2. The Y-chart (a) and the extended Y-chart (b).

that drives the design of the *architecture*. Typically, a designer studies this set of applications, makes some initial calculations, and proposes an initial parameterized architecture. The effectiveness of this architecture is then evaluated by comparing its performance for different values of the parameters. For this *performance analysis*, each application is *mapped* onto the architecture and the performance of each application–architecture–mapping combination is evaluated. The resulting performance numbers may inspire the architecture designer to improve the architecture. The designer may also decide to restructure the application(s) or to modify the mapping of the application(s). These designer actions are denoted by the light bulbs in Figure 2(a).

In this approach, it is assumed that the applications and the architecture are specified in terms of a model of computation and a model of implementation, respectively. For the parallel execution of signal processing nested loop programs (NLPs), a natural model of computation is the *Process Network* model [4,5] which is quite different from the usual imperative model of computation in which the applications are commonly specified. It is, therefore, necessary to provide a compiler that extracts the available parallelism from an application specified as an NLP, say in Matlab, and automatically convert it into a process network specification. We thus extend the Y-chart environment shown in Figure 2(a) to the environment shown in Figure 2(b).

In Section 3 to 5 we focus on the upper right corner in Figure 2(b) which is implemented in our *Compaan* tool set.

3. The Compaan Tool Set

We developed the Compilation of Matlab to Process Networks (Compaan) tool set, which transforms a nested loop program written in Matlab into a process network specification. Compaan does this transformation in a number of steps, shown in Figure 3, leveraging a lot of techniques available in the Systolic Array literature [11]. In Figure 3, a box represents a result and an ellipsoid represents an action or tool.

Compaan starts the transformation by converting a Matlab NLP specification into a *single-assignment code* (SAC) specification. This makes all parallelism available in the original Matlab specification explicit. Next, it derives the *polyhedral reduced dependence*

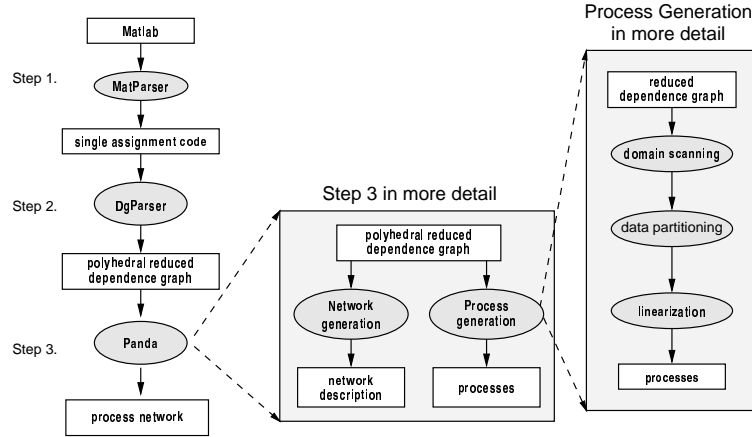


Fig. 3. Compaan consists of three tools that transform a Matlab specification into a process network specification.

graph (PRDG) specification from the SAC. From this PRDG, the network description and the individual processes of the PN are derived. The three steps in Compaan are realized by separate tools: *MatParser*, *DgParser*, and *Panda*, respectively. The last mentioned tool, *Panda*, generates the network and the processes in the process network from the PRDG. The generation of the processes is further decomposed into *domain scanning*, *data partitioning*, and *linearization*. We elaborate on these tools in Section 4 and Section 5.

In the Matlab-to-Process Network compilation, the role of the PRDG is crucial. A PRDG is a compact representation of an NLP's dependence graph. It is a directed graph $G = (V, E)$, where V is a set of *node domains* and E is a set of *edge domains*. The PRDG representation, of the algorithm in Figure 1 is shown in Figure 4*.

3.1. Node Domain

A node domain is characterized by 1) an iteration domain $\mathcal{I} = \{\mathbf{i} = L\mathbf{k} + \mathbf{m} \wedge \mathbf{k} \in \mathcal{P} \cap \mathbb{Z}^n\}$ where $\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$ is a polytope, 2) a function, and 3) a set of port domains. Here, L and A are integral matrices, and \mathbf{i} , \mathbf{k} , \mathbf{m} , and \mathbf{b} are integral vectors. Although L must not be invertible, it is a bijection from $\mathcal{P} \cap \mathbb{Z}^n$ to \mathcal{I} . The function resides in each and every point in the iteration domain. The function takes its arguments from its *input ports* and returns values to its *output ports*. A particular input port or output port belongs to a node domain's *input port domain* (IPD) and *output port domain* (OPD), respectively.

3.2. Edge Domain

An edge domain is an ordered pair (v_i, v_j) of node domains together with an ordered pair (p_i, p_j) of port domains, where p_i is an OPD of v_i and p_j is an IPD of v_j . Unlike the iteration domain of the node domain, the iteration domain of p_i is defined by an affine

*Nodes A , B , and E are source and sink nodes. The corresponding code is not included in the application program in Figure 1

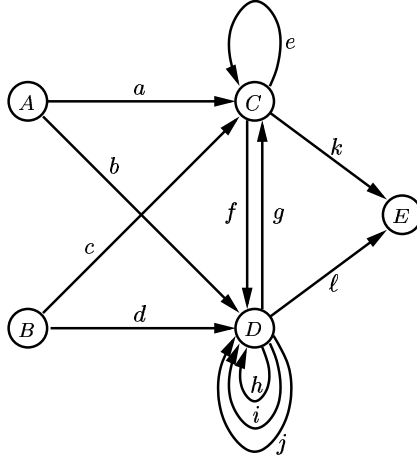


Fig. 4. PRDG of program in Figure 1.

mapping M that is a surjection from IPD to OPD which expresses the data dependency between the output port of the function in v_i and the input port of the function in v_j .

3.3. Example

To illustrate the notion of node and port domains, we show in Figure 5 a part D_0 of node domain D in Figure 4[†]. To be precise, node domain D contains all the $\text{Rot}(\)$ functions of the program in Figure 1. The figure shows the node domain D_0 (a), its iteration domain with the iterators i and j (b), its port domains (c)-(f), and its view as it appears in the PRDG (g) (because we left out the k related domains, only the edges with labels $d, f, i, j,$ and g are depicted).

In (c) and (d), we show IPDs and in (e) and (f), we show OPDs. In (c) we identify two IPDs, ipd_1 and ipd_2 . In (e) we identify two OPDs, opd_1 and opd_2 . The figure shows two edge domains: edge domain i between opd_1 of port domain (e) and ipd_2 of port domain (c) and edge domain j between opd_3 of port domain (f) and ipd_3 of port domain (d).

The PRDG is the intermediate between the given Matlab specification and the required process network specification.

4. MatParser & DgParser

In the path from Matlab to the PRDG, Compaan uses the tools *MatParser* [12,9] and *DgParser* [9]. *MatParser* is an *array dataflow analysis* compiler that finds all parallelism available in affine NLPs written in Matlab using a very aggressive data dependency analysis technique based on integer linear programming [13]. The analysis results in a static representation of the application which enables us to analyse and manipulate it.

MatParser finds whether two variables are dependent on each other, and moreover, at

[†] $\mathcal{I}_0 = \mathcal{I} \cap \{k = k_0\}$.

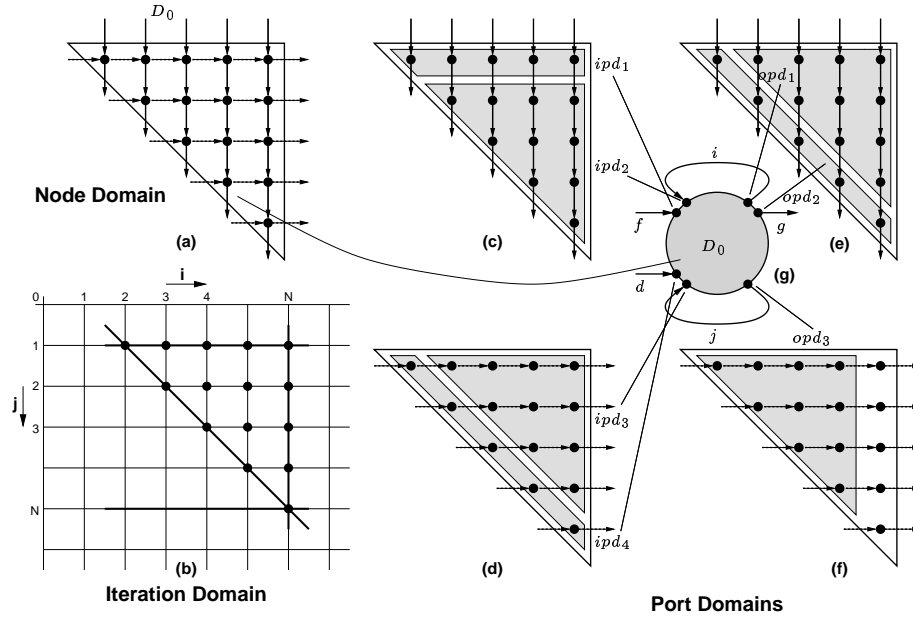


Fig. 5. Part of node domain D and the corresponding port domains.

which iteration. It partitions the iteration spaces defined by the affine control statements, and gives the dependence vector between partitions. For the program given in Figure 1, that only contains for-next control statements, MatParser solves about a hundred parametric integer program problems to find all data-dependencies.

In Figure 6, part of the output of MatParser is shown for the algorithm given in Figure 1. It shows how the iteration space spanned by the for-next iterators k and j is partitioned using if/else statements. Consequently, for different partitions, different data-dependencies may apply. In case of input argument in_0 of function $\text{Vec}()$, either a value previously defined by function Vec should be used (i.e., $r_1(k-1, j)$), defining the data-dependency $M()$, or a value from the original r-matrix (i.e., $r(j, j)$) should be used.

DgParser converts the SAC description into the PRDG description, which is a straightforward conversion. Accordingly, the shape of the node domain is given by the way the for-next loops are defined and the partitioning of the node-domain corresponds with the if/else conditions. The terms ipd and opd used in Figure 6 relate to the IPD and OPD defined in Section 5.

5. Panda

Once DgParser has established a PRDG model of an algorithm, the Panda tool can generate a network description and the individual processes. The network description is straightforward, as it follows the topology of the PRDG. Each node in the PRDG is mapped onto a single process and each edge is mapped onto an unbounded FIFO. In case of Fig-

```

%% Single Assignment Code Generated by MatParser
for k = 1 : 1 : K,
    for j = 1 : 1 : N,

        if k-2>= 0,
            [ in_0 ] = ipd( r_1( k-1, j ) );
        else %% if -k+1 >= 0
            [ in_0 ] = ipd( r( j, j ) );
        end
        if j-2>= 0,
            [ in_1 ] = ipd( x_1( k, j-1, j ) );
        else %% if -j+1 >= 0
            [ in_1 ] = ipd( x( k, j ) );
        end

        [ out_0, out_1, out_2 ] = Vec( in_0, in_1 );

        [ r_1( k, j ) ] = opd( out_0 );
        [ x_1( k, j ) ] = opd( out_1 );
        [ t_1( k, j ) ] = opd( out_2 );
    end
end

```

Fig. 6. Single Assignment Code

ure 4, nodes A , B , C , D , and E will define a process and edges a to ℓ will define an unbounded FIFO.

As shown in Figure 3, the Panda tool divides the generation of a process into three different steps: domain scanning, data partitioning, and linearization. Because the PRDG in Figure 4 is not well suited to illustrate these three steps we use another example in this section. The example used in this section is given in Figure 7 and Figure 8. Figure 7 shows a simple program in Matlab and the single assignment code generated by MatParser. The top part of Figure 8 gives an unfolded view of the PRDG that DgParser generates from the SAC in Figure 7. The bottom part of Figure 8 is the process network that Panda generates from the PRDG. It has two process network nodes, called the producer and the consumer, and an unbounded FIFO channel for the communication between the two. Panda generates the bottom-part of the figure from the top part.

The node domains ND_p and ND_c correspond to the functions $f()$ and $g()$, respectively. The mapping $M()$ represents the data dependency and corresponds with the indexing $(k-1, \ell)$ in line 7 of the SAC. To clearly distinguish between the iteration spaces of the producer and consumer we made a change of variables, i.e., for the ND_p and ND_c we applied the transformations $(k, \ell) \rightarrow (j_2, j_1)$ and $(k, \ell) \rightarrow (i_2, i_1)$, respectively. As a consequence the mapping $M()$ is the function $M : \{j_2 = i_2 - 1 \wedge j_1 = i_1\}$.

5.1. Domain Scanning

The first step in Panda is to scan the node domains by lexicographically ordering the points in the node's iteration domain. Thus, given the iteration domains $\mathcal{J} = \{\mathbf{j} \mid \mathbf{j} = L_p \mathbf{k} + \mathbf{m}_p \wedge \mathbf{k} \in \mathcal{P}_p \cap \mathbb{Z}^2\}$ for ND_p and $\mathcal{I} = \{\mathbf{i} \mid \mathbf{i} = L_c \mathbf{k} + \mathbf{m}_c \wedge \mathbf{k} \in \mathcal{P}_c \cap \mathbb{Z}^2\}$ for ND_c ,

```

1.  %parameter N 4 16;
2.  for k = 0:1:N,
3.      for l = k : 1 : N,
4.          [out_0] = f( );
5.          [x_1( k, l)] = opd( out_0 );
6.          if k-1>= 0,
7.              [in_0] = ipd( x_1(k-
%parameter N 4 16;
for k = 0:1:N,
    for l = k:1:N,
        [x(k,l)] = f();
        [] = g( x(k-1,l) );
    end
end
1,1) );
8.          else %% if -k >= 0
9.              [in_0] = ipd( x(k-
1,1) );
10.         end
11.         [] = g( in_0 );
12.     end
13. end
(a) Original Matlab program.
(b) SAC the MatParser produces.

```

Fig. 7. Original Matlab program and the SAC than MatParser produces from it.

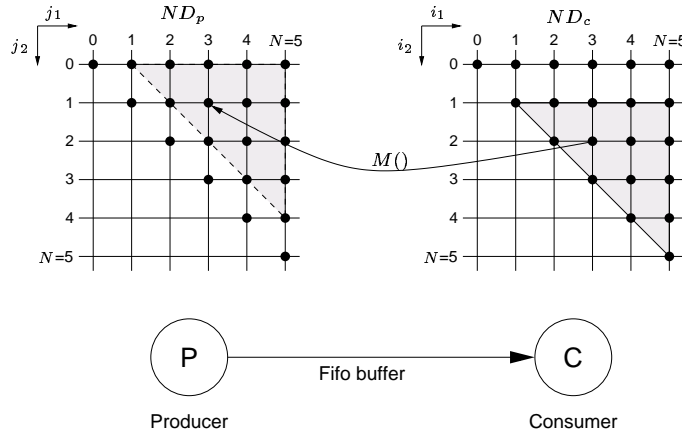


Fig. 8. Mapping the PRDG of Figure 7 onto a Process Network, running example.

respectively, and the iterator scan orders, say (j_2, j_1) for ND_p and (i_1, i_2) for ND_c , the task is to return a lexicographical ordering of the domains in terms of a nested loop. The Fourier-Motzkin procedure [14] is used to accomplish this. Fourier-Motzkin (FM) finds the boundaries of the iterators, given the iterator scan orders. Thus, FM returns for $\text{sort order}(j_2, j_1): \{0 \leq j_2 \leq N, j_2 \leq j_1 \leq N\}$ and for $\text{sort order}(i_1, i_2): \{0 \leq i_1 \leq N, 0 \leq i_2 \leq i_1\}$. The conversion to a nested loop scanning is then straightforward.

5.2. Data Partitioning

MatParser generates a SAC description in which only the IPDs are explicitly specified. This means that the input argument in_0 in Figure 7 is surrounded by if/else statements, while the output value out_0 is not. A consequence of this is that output values can be generated that are never used by some input domain. This problem is illustrated in the

top part of Figure 8; the solid shaded triangle is known explicitly while the dashed shaded triangle is not. The second step in Panda is to make the OPDs explicit.

Making the output port domains explicit is illustrated in Figure 9. It shows two communicating node domains ND_p and ND_c . The tokens produced by port domain P_p of node domain ND_p are to be consumed by port domain P_c of node domain ND_c , as described by the data-dependency with mapping $M()$. Port domain P_p is an OPD and port domain P_c is an IPD. To make P_p explicit, Panda applies $M()$ to IPD P_c .

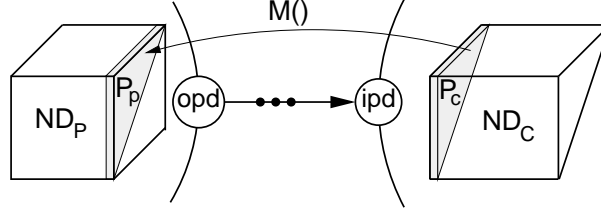


Fig. 9. Making the output port domain explicit.

The procedure goes as follows. Starting point is the relation between IPD and OPD through $M()$. By applying $M()$ to the IPD $\{0 \leq i_1 \leq N \wedge 0 \leq i_2 \leq i_1\}$ of ND_c the OPD of ND_p is found to be $\{0 \leq j_2 \leq N - 1 \wedge j_2 + 1 \leq j_1 \leq N\}$. Comparing these port domains with the respective node domains, it follows that what remains to be checked at run time - to detect the port domains while scanning the node domains - is whether $j_2 + 1 \leq j_1$ in ND_p and $1 \leq i_2$ in ND_c .

5.3. Linearization

The channels between processes are one dimensional FIFO buffers. Therefore, the order in which a consuming process reads tokens from a channel must be the same as the order in which tokens are written onto the channel by the producing process. Of course, the consuming process will in general use the read tokens in a different order (out-of-order consumption). The channel's FIFO and the consumer process's reorder memory are modeled as a single one-dimensional memory m . This is shown in Figure 10.

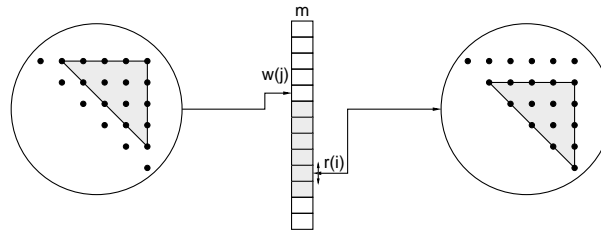


Fig. 10. The producer-to-consumer channel FIFO and the consumer's memory modeled as a single linear memory m .

m is a logical storage structure (LSS) [15], and there is one LSS for each opd/ipd pair. The producer node writes tokens into the LSS in the order given by a *write polynomial* $w(\mathbf{j})$. The consumer node consumes tokens from this LSS in the order given by a *read*

polynomial $r(\mathbf{i})$. The write polynomial follows from a ranking of the OPD of the producer. The ranking follows the order resulting from the domain scanning and data partitioning steps. The read polynomial satisfies $r(\mathbf{i}) = w(M(\mathbf{i}))$, where $M(\mathbf{i})$ is the affine mapping from the IPD of ND_c to the OPD of ND_p . The ranking in ND_p in the running example is shown in Figure 11. The corresponding write polynomial is given as

$$w(j_1, j_2) = -\frac{1}{2}j_2^2 + (N - \frac{1}{2})j_2 + j_1 - 1 \quad (1)$$

The corresponding read polynomial for ND_c is given by

$$\begin{aligned} r(i_1, i_2) &= w(M(i_1, i_2)) = w(i_1, i_2 - 1) \\ &= -\frac{1}{2}i_2^2 + (N + \frac{1}{2})i_2 + i_1 - (N + 1) \end{aligned} \quad (2)$$

The linearization method in Panda relies on methods to count the number of integral points contained within a polytope using so-called *Ehrhart Polynomials* [16]. These methods are implemented in the library *PolyLib* [17].

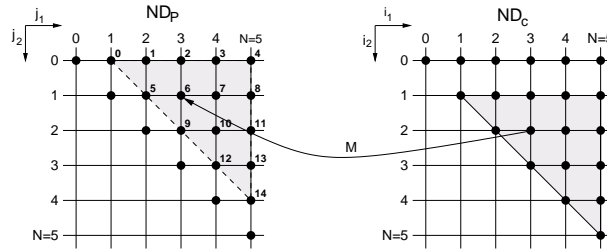


Fig. 11. Ranking of node domains to derive read and write polynomials for addressing the LSS m between producer and consumer.

The three steps *domain scanning*, *data partitioning*, and *linearization* result in a control program as shown in Figure 12 for the running example. As can be seen from the figure,

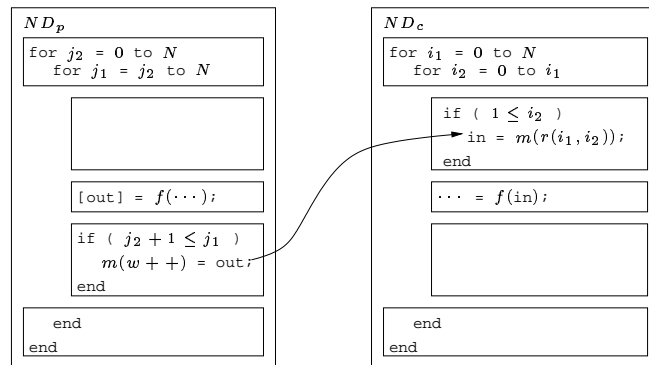


Fig. 12. Control program for writing in-order to the channel and reading - possibly out-of-order - from memory.

ND_p 's \mathbf{j} -loopnest scans the port domain and produces in each and every point the output of the function's firing in these points. If the condition $j_2 + 1 \leq j_1$ is true, then the output values are put in order to the channel. ND_c gets these values from the channel and puts them into a reordering memory. ND_c 's \mathbf{i} -loopnest scans the port domain and if the condition $1 \leq i_2$ is satisfied, then the input value for the function residing in the node domain at the scan point is read from the reordering memory at address $m(r(\mathbf{i})) = m(w(M(\mathbf{i})))$.

6. Conclusion

With the toolset presented in this paper, affine nested loop Matlab programs can be converted to Kahn Process Networks which are to be mapped onto a high level architecture description for simulation and performance analysis on a high level of abstraction. Not discussed in the paper is a tool for performing transformations on the intermediate PRDG specification of the application. Examples of such transformations are *index transformations* and *PRDG unrolling*. All elements of the Compaan tool set are implemented in Java. Part of it is integrated in the Ptolemy II framework (see <http://www.gigascale.org/compaan>).

Acknowledgement

The authors want to thank Paul Lieveise from the Delft University of Technology, Pieter van der Wolf from the Philips Research Laboratories, and Kees Vissers from Trimedia Technologies, inc. for valuable discussions and support. They are also indebted to Vincent Loechner from INRIA for the Polylib guidance and support. The work was supported in part by the universities of Delft and Leiden, in part by Philips Research and in part by the MARCO/DARPA Gigascale Silicon Research Center.

References

- [1] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess, "Prohid, a data-driven multi-processor architecture for high-performance dsp," in *Proc. ED&TC*, Mar. 17-20 1997.
- [2] Edwin Rijpkema, Ed F. Deprettere, and Gerben Hekstra, "A strategy for determining a jacobi specific dataflow processor," in *Proceedings ASAP'97 conference*, July 1997.
- [3] Arthur Abnous and Jan Rabaey, "Ultra-low-power domain-specific multimedia processors," in *VLSI Signal Processing, IX*, 1996, pp. 461-470.
- [4] Gilles Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.
- [5] Edward A. Lee and Thomas M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-799, May 1995.
- [6] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf, "The construction of a retargetable simulator for an architecture template," in *Proceedings of 6th Int. Workshop on Hardware/Software Codesign*, Seattle, Washington, Mar. 15-18 1998.
- [7] Paul Lieveise, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," in *Proceedings of the 1999 IEEE Workshop in Signal Processing Systems*, Taipei, Taiwan, 1999.
- [8] Bart Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures*:

- Methods and Tools*, Ph.D. thesis, Delft University of Technology, Jan. 1999.
- [9] Peter Held, *Functional Design of Data-Flow Networks*, Ph.D. thesis, Dept. EE, Delft University of Technology, May 1996.
 - [10] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proc. ASAP'97*, July 14-16 1997.
 - [11] S.Y. Kung, *VLSI Array Processors*, Prentice Hall Information and System Sciences Series, 1988.
 - [12] Bart Kienhuis, "Matparser: An array dataflow analysis compiler," Tech. Rep. UCB/ERL M00/9, University of California, Berkeley, CA-94720, USA, Feb. 2000.
 - [13] Paul Feautrier, "Parametric integer programming," *Recherche Opérationnelle; Operations Research*, vol. 22, no. 3, pp. 243–268, 1988.
 - [14] C. Ancourt and F. Irigoien, "Scanning polyhedra with DO loops," in *Proc. ACM SIGPLAN '91*, June 1991, pp. 39–50.
 - [15] Alco Looye, Gerben Hekstra, and Ed Deprettere, "Multiport memory and floating point cordic pipeline in jacobium processing element," in *Proceedings SiPS 98 Design and Implementation*, October 1998, pp. 406–426.
 - [16] Ph. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *Journal of VLSI Signal Processing*, vol. 19, pp. 179–194, July 1998.
 - [17] Doran K. Wilde, "A library for doing polyhedral operations," M.S. thesis, Oregon State University, Corvallis, Oregon, Dec 1993, Also published in IRISA technical report PI 785, Rennes, France; Dec, 1993.