

# Translating affine nested-loop programs to Process Networks

Alexandru Turjan, Bart Kienhuis, Ed Depretere  
 Leiden Institute of Advanced Computer Science  
 Leiden, The Netherlands  
 email: aturjan@liacs.nl

Published in Proceedings of the International Conference  
 on Compiler, Architecture, and Synthesis for Embedded Systems, 2004, Washington USA

**Abstract**—New heterogeneous multiprocessor platforms are emerging that are typically composed of loosely coupled components that exchange data using programmable interconnections. The components can be CPUs or DSPs, specialized IP cores, reconfigurable units, or memories. To program such platform, we use the Process Network (PN) model of computation. The localized control and distributed memory are the two key ingredients of a PN allowing us to program the platforms. The localized control matches the loosely coupled components and the distributed memory matches the style of interaction between the components. To obtain applications in a PN format, we have built the *Compaan* compiler that translates affine nested-loop programs into functionally equivalent PNs. In this paper, we describe a novel analytical translation procedure we use in our compiler that is based on integer linear programming. The translation procedure consists of four main steps and we will present each step by describing the main idea involved, followed by a representative example.

## I. INTRODUCTION

Applications envisioned for the next decade in the area of multi-media, imaging, bioinformatics, and signal processing have a high computational demand. To satisfy this demand, new hardware platforms are emerging, referred to as *heterogeneous multiprocessor platforms*. They are typically composed of loosely coupled components that exchange data using programmable interconnections such as a switch matrix or a network on chip. The components can be CPUs or DSPs, specialized IP cores, reconfigurable units, or memories.

Although building such heterogeneous platforms already takes place [21], [31], [26], mapping applications onto them still relies on the ability of a system designer to manually partition the application's memory and control across the platform components [8]. This process is typically performed in an empirical manner, lacking a systematic solution approach. In this process, a designer primarily focuses on the extraction of application independent tasks, the synchronization between the tasks, and on memory management. There are a number of research projects dealing with the automation of the mapping process. For example the PICO project [14], [27] is an effort that aims to automate the mapping of applications onto platforms consisting of VLIW processors and custom nonprogrammable accelerators. Another example is the Atomium [3] project dealing especially with memory issues when

mapping applications onto platforms with distributed memory architectures.

To program heterogeneous multiprocessor platform, we believe that the Process Network (PN) model of computation (MoC) is suitable to cope with the multiprocessor characteristic of the new hardware platforms [25]. The PN is a deterministic MoC that explicitly specifies tasks as processes and distributed memory as FIFO channels [17]. The localized control and distributed memory in a PN are the two key ingredients allowing us to program heterogeneous multiprocessor platforms. The localized control matches the loosely coupled components and the distributed memory matches the style of interaction between the components. However, writing an application in PN format is time consuming and error prone. Therefore, we have built the *Compaan compiler* [16] that translates affine nested-loop programs into functionally equivalent PNs specified in C++ [7] or Java [18] formats. It is also possible to obtain a hardware implementation of the PN using the *Laura* [32] VHDL back-end.

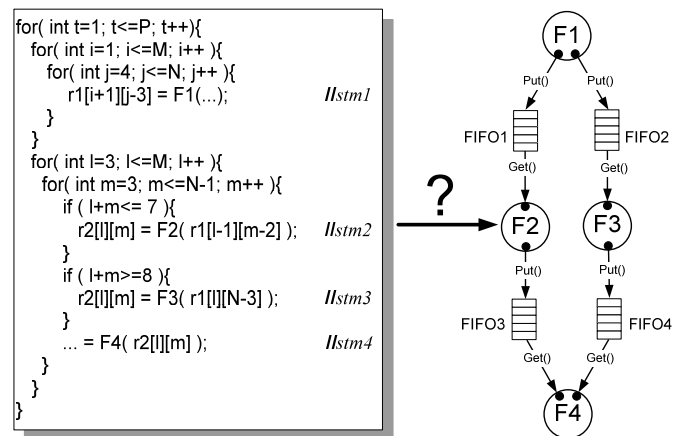


Fig. 1. How to translate an affine nested loop application into a Process Network?

In this paper, we present the analytical procedure our compiler uses to translate affine nested-loop programs into PNs. As we will show, the translation consists of four main steps and we will present each step by describing the main idea involved followed by a representative example. The paper

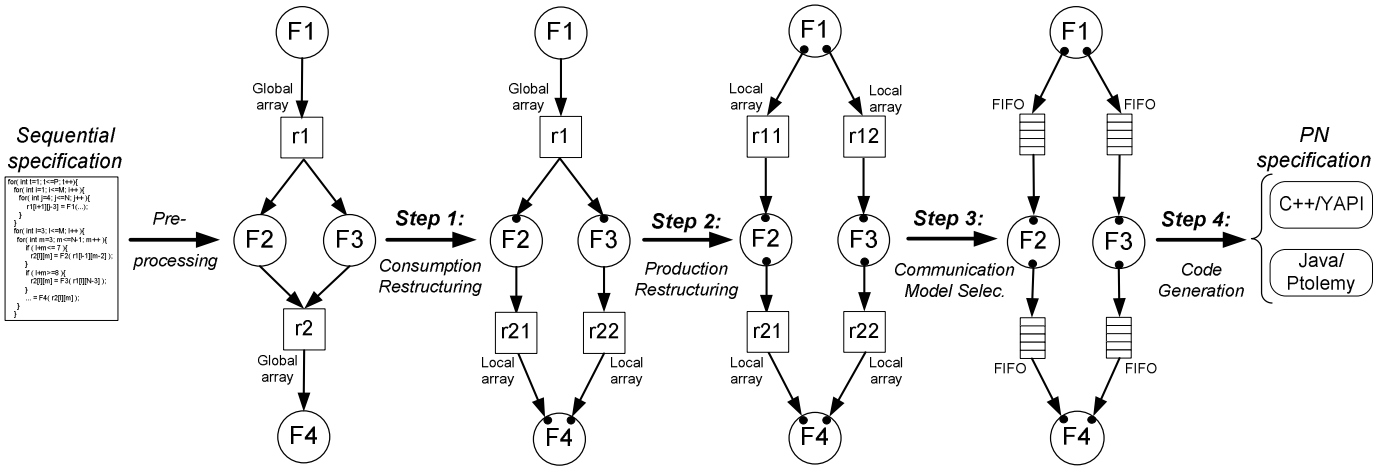


Fig. 2. Deriving a Process Networks in four steps

is organized as follows: In Section II, we give the problem involved in translating an affine nested-loop program to a PN. In Section III, we present a four step approach to do the translation. In Section IV, we give results obtained from running our compiler and in Section V, we conclude the paper.

## II. PROBLEM DEFINITION

The problem we address in this paper refers to the translation of a sequential application to an equivalent PN specification, as shown in Fig 1. The class of applications we consider in this conversion, is confined to nested loops with static control and affine indices [11]. An example of such an application is given in the left side of Fig 1, where each assignment statement is iterated over a convex domain called *iteration space* composed of *iteration points (IPs)* [1]. The iteration spaces can be parameterized by using for-loops with parametric bounds as can be observed in the code. The PN that is generated consists of a number of processes; each process executing one of the assignment statements present in the input program for a number of times. For example, process F1 corresponds to statement *stm1*, process F2 to statement *stm2*, and so on. In the translation from an affine nested loop program to a PN, two problems are involved. First, the computation carried out by a sequential application in a single process needs to be distributed into a number of separate computational processes. Secondly, the global memory arrays (e.g., *r1* and *r2*) used for data storage need to be transformed to dedicated FIFO buffers that are accessed using a blocking *get* primitive, providing in this way a simple inter-process synchronization mechanism.

The way we approach the translation problem originates from the work done by Held [13] and Rijpkema [24]. Held tried to obtain a systolic array for the same class of applications we consider. Rijpkema was the first one who formulated the translation to PNs. He partitioned the problem into three steps as realized in a tool called *Panda* [16]. In these steps, he made use of the Ehrhart theory [9], [4]. Due to the complexity and implementation limitations of this theory [29], the proposed procedure was validated only for a limited class

of input algorithms. In this paper, we present a new solution to the translation problem given in Fig 1 that uses Integer Linear Programming (ILP). As a consequence, our translation approach fully converts the class of static nested-loop program with affine indices.

## III. SOLUTION

The conversion from an application to a PN takes place gradually in a number of steps guided by the idea of localizing the control and distributing the memory as shown in Fig 2. As a result of a *Preprocessing* step, the initial sequential specification is converted to a network representation where all the executions of one assignment statement are collapsed into a single process.

This network represents the input of the first step, the *Consumption Restructuring*. During this step, we restructure the data consumption, i.e., each array used for storing data generated by different producer processes is replaced by a number of separated memory arrays; one for each producer process. In the second step, *Production Restructuring*, we restructure the data production, i.e., each array used for storing data consumed by several consumer processes is replaced by a number of separated memory arrays; one for each consumer process. After performing the first two steps a distinct piece of memory is put between a producer and consumer process. This forms an instance of the classical *producer/consumer (P/C) pair*. Depending on the order data is produced and consumed in a P/C pair, different types of communication mechanisms should be employed with adequate synchronization policies to derive a valid PN. This is done in the third step called *Communication Model Selection*. Using the information obtained in first three steps, a PN with autonomously running processes communicating data over FIFO channels is obtained as Java or C++ code in the last step of our approach called *Code Generation*.

The network obtain after the Preprocessing step does not reveal any degree of parallelism. This it just a partitioned representation of the application code given in Fig 1. The topology of this network resembles the Reduce Dependence

Graph [6] of the application. Each circle from the left part of Fig 2 represents a process iterating one of the assignment statements over the same iteration space as the statement is iterated in the original code. The processes are still executed one at the time following the same global schedule in which the correspondent assignment statements are executed in the original code.

### A. Step1 - Consumption Restructuring

In the Consumption Restructuring step, data consumption is restructured such that each producer process can store data into a separate memory array. Hence, no two producer processes write data into the same array. This transformation is visualized in Fig 3, where array  $r2$  is replaced by two different arrays  $r21$  and  $r22$ . Due to the restructuring, the process  $F4$  now has to decide at each execution whether to read data from  $r21$ , or  $r22$ . Consequently, the iteration space of process  $F4$  gets partitioned into two subdomains. Each subdomain represents what we call an *Input Port Domain* (IPD). Therefore, IPD1 contains the IPs at which process  $F4$  reads data produced by process  $F2$  and the other one, IPD2, contains the IPs at which process  $F4$  reads data produced by process  $F3$ . Graphically, the IPDs of a process are visualized in Fig 2 as black spots located at the end of a consumer process incoming edge. The partitioning in IPDs is done by adding linear inequalities to the domain of  $F4$  as shown in the code in Fig 3.

1) *Approach*:: To derive the inequalities of the IPD that partition the consumer domain, we first identify groups of producer processes that are writing data into the same memory array. Let  $S_r$  be the set of all the processes  $P_i^r$  that write data into the memory array  $r$  and  $D_r$  the set of all processes  $C_j^r$  that read data from  $r$ . For each process  $P_i^r$ , we replace the writing in array  $r$  with a write into a separate array  $r_i$ . To maintain a correct execution, the corresponding processes  $C_j^r$  have to consume data from the new memory arrays  $r_i$ . Therefore, we have to be able to connect the consumption of a data token with its production. To solve this problem, we make use of exact data dependence analysis [11], [22], [19]. By performing the dependence analysis, we get an affine dependency function together with the domain where this function is valid. This domain actually is an IPD. Each IPD represents an integral union of parameterized polytopes containing all IPs at which the input argument of the assignment statement embedded in the process is being produced by one process. Without loss of generality, we will assume that each IPD is represented by only one integral parameterized polytope of dimension  $k$ ,  $IPD = \mathcal{C}(N) \cap \mathbb{Z}^k$ . Thus, each P/C pair is uniquely represented by a polytope  $\mathcal{C}(N)$  together with an affine dependency function  $f$  represented by an integral matrix  $M$ , and an offset vector  $O$ , i.e.,  $f(x) = Mx + O$ .

2) *Example*:: Consider the code given in Fig 1, where the statements  $stm2$  and  $stm3$  are responsible for writing data into a 2-d array  $r2$  from where statement  $stm4$  consumes data. In the network representation of the original application, we identify the P/C pairs of processes  $PC_1 = (F2, F4)$  and  $PC_2 = (F3, F4)$ , each of them communicating data via the

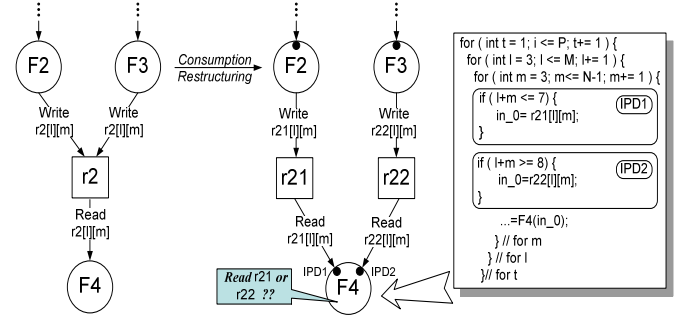


Fig. 3. Consumption restructuring - need for dependences analysis

global array  $r2$ . We replace in each P/C pair the write into array  $r2$  at location  $r2[l][m]$ , with a write into array  $r21$  at location  $r21[l][m]$  and respectively into array  $r22$  at location  $r22[l][m]$ . As a consequence, process  $F2$  and  $F3$  will write data into separate memory arrays as shown in the right-hand side of Fig 3.

To keep the execution of the network correct, we have to find at each execution of process  $F4$  the location in  $r21$  or  $r22$  containing the appropriate input data. This correspondence is obtained using the data-dependency functions corresponding to the P/C pairs of statements  $(stm2, stm4)$  and  $(stm3, stm4)$ . In case of pair  $PC_1$ , we find the dependency function  $f_{PC_1}(t, l, m) = (t, l, m)$  being valid on the input port domain  $IPD_1 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l+m \leq 7\}$ . Hence, at an IP  $(t, l, m)$  belonging to  $IPD_1$ , the process  $F4$  consumes data produced by process  $F2$  that is stored in  $r21[l][m]$ . In case of  $PC_2$ , by doing a similar analysis, we find the dependency function  $f_{PC_2}(t, l, m) = (t, l, m)$  valid on  $IPD_2 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 8 \leq l+m\}$ , such that process  $F4$  has to read data stored in  $r22[l][m]$ . At this step we derive also the dependency functions corresponding to the PC pairs  $PC_3 = (F1, F2)$  and  $PC_4 = (F1, F3)$ . They are  $f_{PC_3}(t, l, m) = (t, l-2, m-1)$  valid on  $IPD_1 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, l+m \leq 7\}$  and  $f_{PC_4}(t, l, m, n) = (t, l-1, N)$  valid on  $IPD_1 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N-1, 8 \leq l+m\}$ . These functions will be used in the steps 2 and 3 of our compilation process.  $\square$

### B. Step2 - Production Restructuring

In the Production Restructuring step, we replace the memory arrays that are accessed by different consumer processes with a separate array for each consumer process. This transformation is visualized in Fig 4, where array  $r1$  is replaced by two different arrays  $r11$  and  $r12$ . Due to the restructuring, process  $F1$  now has to decide at each execution whether to write data to  $r11$ ,  $r12$ , both arrays, or even none of the arrays. The restructuring will partition the iteration space of process  $F1$  into two subdomains. Each subdomain represents what we call an *Output Port Domain* (OPD). Therefore, OPD1 contains the IPs at which process  $F1$  writes data consumed by process  $F2$  and the other one, OPD2, contains the IPs at which process  $F1$  writes data consumed by process  $F3$ . Graphically, the OPDs

of a process are visualized in Fig 2 as black spots located at the beginning of an outgoing edges of a Producer process. The partitioning in OPDs is done by adding linear inequalities to the domain of  $F1$  as shown in the code given in Figure 4.

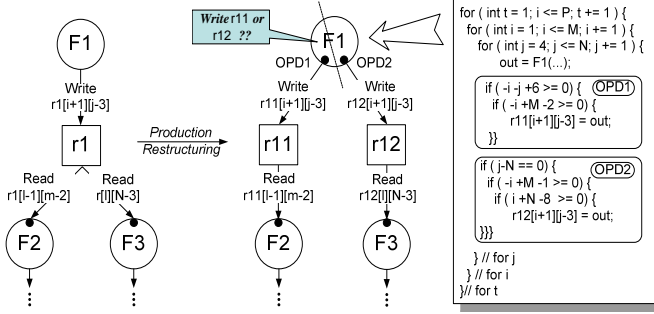


Fig. 4. Production Restructuring

1) *Approach*:: To derive the inequalities of the OPDs that partitions the producer domain, we first identify groups of consumer processes that are reading data from the same memory array. Let  $D_r$  be the set made of all the consumer processes  $C_i^r$  that are reading data from the same memory array  $r$ . For each process  $C_i^r$ , we replace the read of data from the global array  $r$  with a read from a separate array  $r_i$ . Due to the Consumption Restructuring step, there is only one process  $P^r$  that writes data into  $r$ . To have a correct execution, the producer processes  $P^r$  has to decide at each IP  $y$  what are the proper storage arrays  $r_i$  where data has to be written. Finding for a producer IP  $y$  the appropriate storage arrays, is equivalent with deciding whether  $y$  belongs to the following set:

$$OPD = f(\mathcal{C}(N) \cap \mathbb{Z}^k) = \{p \mid p = f(x), x \in (\mathcal{C}(N) \cap \mathbb{Z}^k)\}, \quad (1)$$

where  $f$  is the dependency function and  $\mathcal{C}(N)$  is the parametrized consumer IPD. Finding whether  $y$  belongs to  $OPD$  can be expressed as the solution of the following parametric integer linear programming (PIP) problem [10], with variable  $x_c$  and parameter  $y_c$ :

$$\text{subject to: } x_c \in \mathcal{C}(N), \quad (c1)$$

$$y_p = f(x_c), \quad (c2) \quad (2)$$

$$\text{objective: } c_m = \min_{\text{lex}} \{x_c(y_p)\},$$

where condition (c1) specifies that the problem domain is given by the polytope  $\mathcal{C}(N)$ , and (c2) imposes that the problem should include only the integer points  $y_p$  for which a consumer point  $x_c$  exists. Although we are interested only whether an integral solution exists or not, we choose as objective the lexico-minimal function. This allows us to gather additional information which is used in the Code Generation step to optimize the network memory management. As shown in [11], the solution of the presented problem is a multistage

conditional expression:

$$\begin{aligned} & \text{if } (y_p \in \mathcal{D}_1), \\ & \quad \text{then } x_c = T_1(y_p), \\ & \text{else if } (y_p \in \mathcal{D}_2), \\ & \quad \text{then } x_c = T_2(y_p), \\ & \quad \vdots \\ & \text{else if } (y_p \in \mathcal{D}_n), \\ & \quad \text{then } x_c = T_n(y_p). \end{aligned}$$

where  $\mathcal{D}_1, \dots, \mathcal{D}_n$  are disjoint parameterized polytopes subparts of  $P$  and  $T_1, \dots, T_n$  are affine transformations. Some of the solution branches can have an empty statement represented as  $T_i = \perp$ . This corresponds to the case when  $f$  is not a surjective function. Only when a producer IP belongs to a non-empty branch, data is consumed by a consumer process and it has to be stored into a memory array. Although in this step, the expressions of the  $T_i$  functions do not serve a purpose, they are used in the Code Generation step for the lifetime analysis of tokens to optimize the network memory management. Due to the restructuring, an  $OPD$  is the union of the domains expressed by the non-empty tree branches:  $OPD = \cup_{i=1}^n \mathcal{D}_i$ ,  $T_i \neq \perp$ . It is easy to observe that this formulation of an  $OPD$  is equivalent to the one given in Equation 1.

2) *Example*:: In case of process  $F1$ , we have to make explicit two OPDs, namely  $OPD1$  consisting of the iterations at which data has to be loaded into  $r11$  and  $OPD2$  consisting of the IPs at which data has to be loaded into  $r12$ . Since we have two P/C pairs, the following two PIP problems (corresponding to  $PC_3$  and to  $PC_4$ ) have to be solved:

$$\begin{aligned} \text{Problem } PC3: \\ \text{subject to: } & \begin{cases} 1 \leq t_p \leq P, & 3 \leq m_c \leq N-1, & (c1) \\ 3 \leq l_c \leq M, & l_c + m_c \leq 7, & (c2) \\ (t_p, l_c, j_p) = (t_c, l_c - 2, m_c + 1), & (c3) \end{cases} \\ \text{objective: } & \min_{\text{lex}} (t_c, l_c, m_c). \\ \text{Problem } PC4: \\ \text{subject to: } & \begin{cases} 1 \leq t_p \leq P, & 3 \leq m_c \leq N-1, & (c1) \\ 3 \leq l_c \leq M, & 8 \leq l_c + m_c, & (c2) \\ (t_p, l_c, j_p) = (t_c, l_c - 1, N), & (c3) \end{cases} \\ \text{objective: } & \min_{\text{lex}} (t_c, l_c, m_c). \end{aligned}$$

As shown in [10], [23], the two ILP problems can be solved using algorithms like Lexicographical Dual Simplex or Fourier-Motzkin Elimination. As a result we get the following two solution trees  $ST_1$  and  $ST_2$ , composed of statements expressed in the coordinates of the iteration space of process  $F1$ :

Solution Tree  $ST_1$ :

$$\begin{aligned} & \text{if } (1 \leq t_p \leq P) \{ \\ & \quad \text{if } (1 \leq i_p \leq M-2) \{ \\ & \quad \quad \text{if } (4 \leq j_p \leq N) \{ \\ & \quad \quad \quad \text{if } (i_p + j_p \leq 6) \{ \\ \text{Sol: } & \quad (t_c, l_c, m_c) = (t_p, i_p + 2, j_p - 1); \\ & \quad \} \\ & \quad \} \\ & \quad \} \\ & \} \end{aligned}$$

Solution Tree  $ST_2$ :

$$\begin{aligned} & \text{if } (1 \leq t_p \leq P) \{ \\ & \quad \text{if } (1 \leq i_p \leq M-1) \{ \\ & \quad \quad \text{if } (j - N == 0) \{ \\ & \quad \quad \quad \text{if } (i_p + N - 8 \geq 0) \{ \\ & \quad \quad \quad \quad \text{if } (2 \leq i_p \leq 4) \{ \\ \text{Sol1: } & \quad (t_c, l_c, m_c) = (t_p, i_p + 1, -i_p + 7); \\ & \quad \quad \text{if } (5 \leq i_p) \{ \\ \text{Sol2: } & \quad (t_c, l_c, m_c) = (t_p, i_p + 1, 3); \\ & \quad \quad \} \\ & \quad \quad \} \\ & \quad \} \\ & \} \} \end{aligned}$$

The two trees from above partition the iteration space of process  $F1$ . While initially the data produced by  $F1$  was always written at  $r[i+1][j-3]$ , now depending on the constraints specified by the branches of  $ST1$  and  $ST2$ , the data is written into  $r12[i+1][j-3]$  and/or into  $r12[i+1][j-3]$  as shown in Fig 4. Observe that the solution tree of  $ST2$  has two different solution, corresponding to two disjoint domains:  $D_1 = \{(t_p, i_p, j_p) \in \mathbb{Z}^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8, 2 \leq i_p \leq 4\}$  and  $D_2 = \{(t_p, i_p, j_p) \in \mathbb{Z}^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8, 5 \leq i_p\}$ . This we interpret as follows: if  $ip_1 = (t_p, i_p, j_p)$  is a producer IP belonging to  $D_1$  at which  $F_1$  produces the token  $t$ , then  $(t_p, i_p + 1, -i_p + 7)$  represents the first (lexicographically smallest) consumer IP that consumes the token  $t$ . Similarly, if  $ip_2$  is a producer iteration point belonging to  $D_2$  then the first consumer IP that consumes it is  $(t_p, -i_p + 1, 3)$ . However, from the point of view of the distribution of the data the information regarding different first consumption is irrelevant here i.e., we are interested only whether a produced token has to be submitted or not. Therefore, as shown by Fig 4 the two disjoint domains  $D_1$  and  $D_2$  represent the output port domain  $OPD_2 = D_1 \cup D_2 = \{(t_p, i_p, j_p) \in \mathbb{Z}^3 \mid 1 \leq t_p \leq P, 1 \leq i_p \leq M-1, j_p = N, 0 \leq i_p + N - 8\}$ .  $\square$

### C. Step3 - Communication Model Selection

After performing the Consumption and Production Restructuring, the original application has been partitioned into separate tasks in which P/C pairs communicate data over dedicated memory arrays. In the *Communication Model Selection* step, we investigate the communication characteristics of each P/C pair in order to replace the memory array with a FIFO based communication structure. As result of this step, a PN with bounded memory execution is obtained. This is because a FIFO size equal to the number of IPs included into the corresponding OPD will be enough to avoid the appearance of network deadlocks. However, the size of the FIFO can be decreased using techniques allowing us to find a good balance between memory space and inter-process parallelism [20], [2].

1) *Approach*:: There are four communication types for a P/C pair. These four types of communication are given in Figure 5. They result from the *ordering* of the iterations at the Producer and the Consumer processes and the existence of *multiplicity* for a given token, which means that a token that is sent by the Producer is read more than once at the Consumer side. We define in a formal way ordering and multiplicity as follows:

**Definition 1** A P/C pair is **in-order** iff the dependency function  $f : (\mathcal{C} \cap Z^k) \rightarrow P$  preserves the order, i.e., every two Consumer iteration points  $x_1 \prec x_2$  are mapped onto two Producer iteration points  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$  such that  $y_1 \preceq y_2$ . If a P/C pair is not in order we call it **out-of-order**.

**Definition 2** A P/C pair is **without multiplicity** iff the mapping  $f : (\mathcal{C} \cap Z^k) \rightarrow P$  is injective, i.e.,  $\forall x_1, x_2 \in \mathcal{C} \cap$

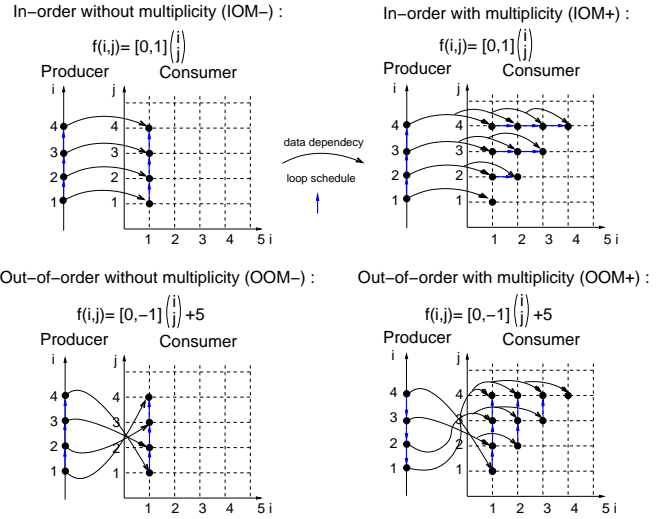


Fig. 5. Four possible types of P/C data-flow graphs

$Z^k$  s.t.  $x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$ . Otherwise we say that the P/C pair is **with multiplicity**.

According to these definitions, an arbitrary P/C pair belongs to one of four disjoint classes: *in-order without multiplicity (IOM-)*, *in-order with multiplicity (IOM+)*, *out-of-order without multiplicity (OOM-)*, and *out-of-order with multiplicity (OOM+)*.

To determine the communication pattern of an arbitrary P/C pair, we need to identify to which of the four classes the P/C data-flow graph belongs. For that purpose, we introduce two tests. The *Reordering Test* determines if a P/C pair is in-order and the *Multiplicity Test* determines if a P/C pair is with multiplicity. Based on these two tests, an arbitrary P/C pair is classified to one of the four categories. These two tests can be formulated and solved using ILP. Consider again an arbitrary P/C pair  $PC$  represented by a parameterized IPD  $\mathcal{C}(N)$  and a dependency function  $f$ . According to Definition 1, a P/C pair is out-of-order, if there exist two Consumer IPs  $x, y$  (as given by conditions (c1) and (c2)), such that  $x \prec y$  (c3) and  $f(x) \prec f(y)$  (c4). These four conditions form the *Reordering Problem (RP)*. If a solution exists for the RP, it means that a P/C pair is out-of-order so the RT is true. Otherwise the P/C pair is in-order.

$$\text{RP : } \begin{cases} x \in (\mathcal{C}(N) \cap Z^k), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^k), & (c2) \\ x \prec y, & (c3) \\ f(y) \prec f(x). & (c4) \end{cases}$$

According to Definition 2, a P/C pair has multiplicity if two different Consumer points  $x$  and  $y$  exists as given by conditions (c1), (c2) and (c3), such that they consume one and the same token from the Producer as given by condition (c4). The four conditions form the *Multiplicity Problem (MP)*. If a solution exists for the MP then the MT is true such that

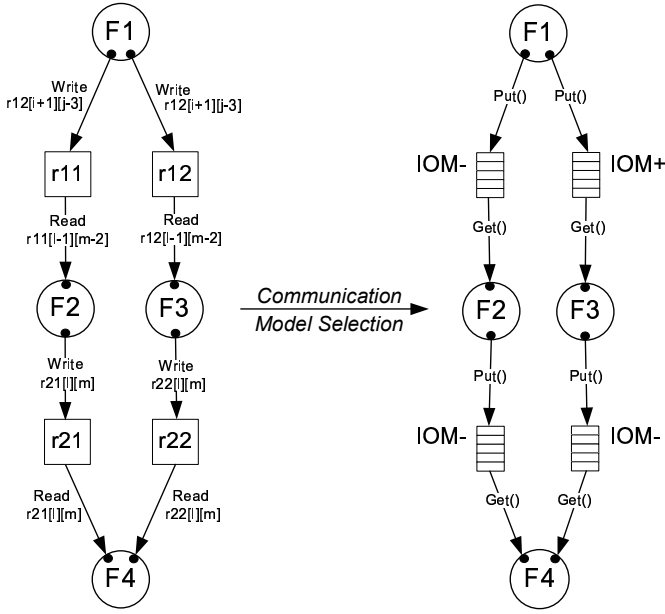


Fig. 6. Communication Model Selection

then the P/C pair is with multiplicity.

$$\text{MP} : \begin{cases} x \in (\mathcal{C}(N) \cap Z^k), & (c1) \\ y \in (\mathcal{C}(N) \cap Z^k), & (c2) \\ x \neq y, & (c3) \\ f(x) = f(y). & (c4) \end{cases}$$

Both the Multiplicity Test and the Reordering Test are so called *Existence tests* as we only need to determine whether MP and RP have at least an integral solution. The procedure to determine if a domain contains at least a single integer point is what we call the *Empty Domain Test* (ET). To realize the ET we make use of the *Omega Test* as provided by PIP or Omega libraries [22], [10]. The ET is an ILP test and requires systems of linear constraints. Both *MP* and *RP* contain non-linear constraints (see for example conditions (c3) in both problems), but using the lexicographic order, we can decompose them into subsets of linear constraints ( $MP_i$  respectively  $RP_j$ ) onto which ET can be applied:  $MT(MP) = \bigvee_i ET(MP_i)$  and  $RT(RP) = \bigvee_j ET(RP_j)$ . In case of the RP, the lexicographical order operator  $\prec$  is decomposed into subsets of linear constraints. On each subset, the ET needs to be applied. In case of the MP, the negation is the non-linear operator. The negation can be rewritten to two inequalities, as  $x \neq y \Leftrightarrow y \prec x \vee x \prec y$ , where we use again the decomposition of the lexicographical operator to obtain linear constraints.

2) *Example*:: Let us analyze how the presented tests are used for deciding the type of an arbitrary P/C pair. Due to space constraints, we present only how the MT applies in case of  $PC_4 = (F1, F3)$ . For this purpose we verify whether the domain specified by the constraints given in  $MP_{PC_4}$  contains integer points. As you can see in  $MP_{PC_4}$ , all the constraints are linear inequalities excepting those specified by the condition (c3). Because  $x$  and  $y$  are arbitrarily points from

$$\text{MP}_{PC_4} : \begin{cases} 1 \leq x_t \leq P, & (c1) & 1 \leq y_t \leq P, & (c2) \\ 1 \leq x_l \leq M, & & 1 \leq y_l \leq M, & \\ 4 \leq x_m \leq N, & & 4 \leq y_m \leq N, & \\ 8 \leq x_l + x_m, & & 8 \leq y_l + y_m, & \\ (x_t, x_l, x_m) \neq (y_t, y_l, y_m) & & (c3) & \\ (x_t, x_l - 1, N) = (y_t, y_l - 1, N) & & (c4) & \end{cases}$$

$\mathcal{C}(N)$  by using the lexicographical order the condition (c3) is decomposed as the following set of linear conditions:

$$(x_t, x_l, x_m, x_n) \neq (y_t, y_l, y_m, y_n) \equiv \begin{cases} x_t < y_t \vee & (c3^1) \\ (x_t = y_t, x_l < y_l) \vee & (c3^2) \\ (x_t = y_t, x_l = y_l, x_m < y_m) \vee & (c3^3) \end{cases}$$

This leads to three instances of the MP. If one of these systems has a solution, multiplicity is involved, which is the case. The system made of conditions (c1), (c2), (c3<sup>3</sup>), (c4) has a solution. This can be verified by looking, for example, to the points  $P1 = (t, l, v)$  and  $P2 = (t, l, w)$  with  $v \neq w$ . Both points are mapped to the same point  $(t, l - 1, N)$  at the Producer side. By applying the multiplicity and reordering tests to the 4 P/C pairs in our example, we find that  $PC_1, PC_2, PC_3$  are of type IOM- and that  $PC_4$  is of type IOM+ as shown in Fig 6.  $\square$

#### D. Step4 - Code Generation

In the first three steps, we have created a PN model which consists of a topology, the iteration spaces of the processes, the IPDs and OPDs, and the types of the channels. In the Code Generation step, a software representation is derived for the PN model. The iteration spaces are converted to for-statements by making use of Fourier-Motzkin Elimination [5]. The topology, IPDs, and OPDs are transformed into components like threads and sets of for and if statements with linear expressions. For the discussed components, equivalent implementation exists in the YAPI environment, as C++ [7], or in the PN-domain of the Ptolemy Framework which is based on Java [18]. In this step, we also take advantage of the classification done in the Communication Model Selection step, to implement an optimal communication structure for each P/C pair.

1) *Approach*:: To derive a software description of the PN takes place in two steps. In the first step, the network processes are derived. Each iteration space of a process, which is represented by a matrix, is translated to a nested for-loop representation. Furthermore, each IPD and OPD is translated from its matrix representation to a structure of if-statements that is inserted in the appropriate processes. In the second step, the network communication structure is derived for each P/C pair. Based on the type of the P/C pair, we realize the communication in the follow way:

- **IOM-** Using only a FIFO buffer that is accessed using a get and put primitive.
- **IOM+** Using a FIFO buffer that is accessed using a get and put primitive. However additional control is added

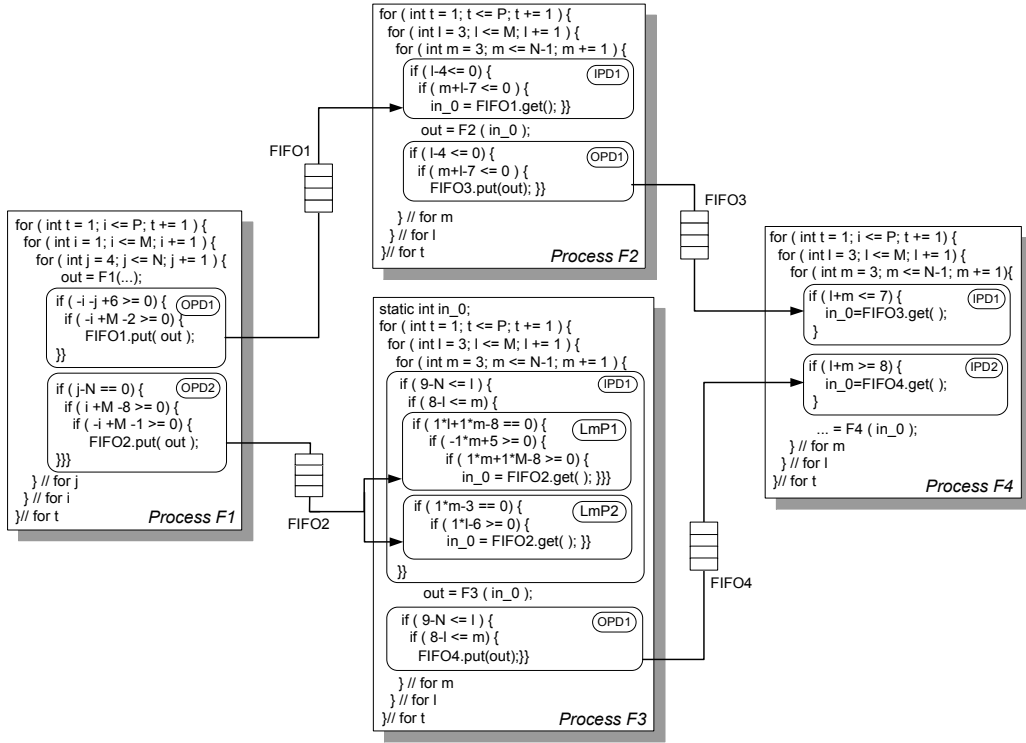


Fig. 7. The final network implementation

to determine the life-time of a token to account for the multiplicity of tokens.

- **OOM-** Using a FIFO buffer that is accessed using a get and put primitive, but at the Consumer process we add private reordering memory and a controller to perform the reordering. Since multiplicity is not involved, each time the controller accesses the reordering memory for reading data, the corresponding memory location can be immediately released.
- **OOM+** Using a FIFO buffer that is accessed using a get and put primitive, but at the Consumer process we add private reordering memory and a controller to perform the reordering and additional control to keep track of the life-time of a token. If the life-time of the token has come to an end, the life-time control releases the memory location hold by the token in the reordering memory.

The implementations for the different types increase in their complexity from IOM- to OOM+. The implementation of IOM- and IOM+ are closely related, except that in IOM- additional control is needed to know when to read data from the FIFO. The implementation of OOM- and OOM+ requires additional reordering memory and a reorder controller. Of the four models identified, OOM+ is the most expensive communication structure to be realized. It is also the generic communication structure since it subsumes all three other structures.

To perform a compile time lifetime analysis of data communicated between Producer and Consumer processes, for communication type IOM+, we make use of what we call the *Lexicographically minimal Preimage* (LmP) [28]. The LmP

maps the domains  $\mathcal{D}_1, \dots, \mathcal{D}_n$  presented in the solution tree presented in Section III-B into the Consumer domain using the non-empty functions  $T_1 \dots T_n$ . These transformations are the solution to the minimization problem given in Equation 2, with as objective to find the lexicographical minimal. Hence, an iteration  $y \in T_1(\mathcal{D}_1)$  is therefore the lexicographically minimal IP that consumes the token produced by  $f(y)$ . This means that  $y$  is a point at which a new token has to be read from a FIFO. Once the token is read and removed from the FIFO, it can be reused as many times as needed, until the next  $y'$  is found that indicated that a new token is to be read.

The opposite of the LmP is the *Lexicographically Maximal Preimage*. This identifies the last consumer IP which uses a certain input data token. For communication type OOM+, where the tokens are stored in a reordering memory, the Lexicographically Maximal Preimage indicates when a memory location can be released allowing us to minimize the size of the reordering memory.

2) *Example*:: In the example, we focus only on the implementation of the communication types IOM+ and IOM-. In case of pair  $PC_1, PC_2, PC_3$ , we replace the static arrays  $r11, r21$  and  $r22$  with a FIFO buffer. Observe that the absolute addressing performed on the arrays is now replaced by a relative addressing using put and get primitives. In case of  $PC_4$ , we replace static array  $r12$  by a FIFO buffer, but we also need to take into account the life-time of the tokens flowing over the FIFO due to multiplicity. To find the moment a process can read a token from FIFO2. we use the LmP. We map the domain represented by  $OPD_2$  through their correspondent solution functions. Hence, we map the domain  $\mathcal{D}_1$  through affine mapping  $(t_p, i_p + 1, -i_p + 7)$  and we get

$LmP_1 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N - 1, l + m = 8, n = 3, m \leq 5\}$ . Similarly, we map domain  $D_2$  through affine mapping  $(t_p, -i_p + 1, 3, 3)$  and we get domain  $LmP_2 = \{(t, l, m) \in \mathbb{Z}^3 \mid 1 \leq t \leq P, 3 \leq l \leq M, 3 \leq m \leq N - 1, 3 \leq m \leq M, m = 3, 6 \leq l \leq 9 - N\}$ . Once  $LmP_1$  and  $LmP_2$  have been derived, we can simplify them in context of the correspondent process IPD by removing constraints in common with the constraints describing the IPD.

The pseudo code for the PN is shown in Fig 7. It shows the way the four processes are implemented. It also shows how the IPDs and OPDs derived in the various steps, are transformed into if-statement using linear expressions. In case of Process  $F3$ , we need to implement the constraints that take care of the life-time of tokens into account. The simplified  $LmP_1$  and  $LmP_2$  are converted to if-statements and inserted to the IPDs of Process  $F3$ . If conditions  $LmP_1$  or  $LmP_2$  hold, a token is read from the FIFO and is reused as many times as needed before the  $LmPs$  indicate that the next token needs to be read.  $\square$

#### IV. IMPLEMENTATION AND RESULTS

The steps presented in Fig 2 are implemented in the tool chain shown in Fig 8. The first tool, called MatParser [15], performs an exact data-dependence analysis. This tool implements the Consumption Restructuring step. The *Process Network Generator* tool, or *PNGen*, implements the remaining three steps and generates a PN description. PNGen replaces the Panda tool in Compaan. The user can choose the PN to be generated in C++ or in Java. The generated code allows us to simulate the PN and to verify that the PN is equivalent to the original sequential program. It is also possible to generate hardware for a PN. The *Laura* tool [32] transforms the network generated by *PNGen* into an equivalent VHDL description that can be synthesized and mapped on an FPGA platform. The four transformation steps make extensive use of polyhedron manipulations, matrix decompositions, and integer linear programming. We relay for these operations within MatParser and PNGen on existing libraries, like PolyLib [30], Pip [10], and Omega [22].

In Table I, we present some quantitative characteristics obtained from compiling 7 applications, of which the JPEG case is described separately in [25] and the QR case in [12]. For each application, we have given the number of lines of the original sequential representation, the compilation time required on a Pentium III processor, and the number of processes and channels generated. We also show how the P/C pairs are classified to the four types. Based on this data, we observed that in approximately 90% of the P/C pairs a communication structure based on a FIFO buffer is sufficient. In remaining 10% of the cases, we need to realize a reordering at the Consumer process, using extra memory and a reordering controller.

#### V. CONCLUSION

This paper describes a solution to convert the complete class of static affine nested loop programs into equivalent PN representations equivalent using Integer Linear Programming.

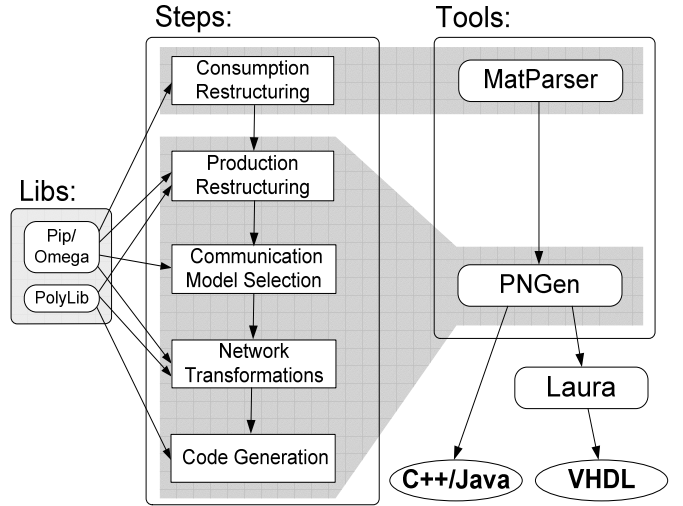


Fig. 8. Compiler organization

The approach is analytical; there is not a single heuristic involved. We have shown that the conversion problem can be divided into 4 steps. For each step, we presented the main idea of the step, how to realize the step, and how it applies to a running example.

All the steps and techniques presented have been implemented in software in the Compaan tool chain. Actually, all the examples given in this paper are generated by this compiler. We also showed the results we get from running Compaan on a set of 7 applications from the area of signal and image processing. The Compaan compiler put us in a great position to program heterogeneous multiprocessor platforms. Using PNGen, we obtain software implementations for PNs that can be mapped and executed on CPUs or DSPs. On the other hand, using Laura, we can also obtain hardware implementations for PNs making use of dedicated IP cores and reconfigurable hardware. An arbitrary mix between hardware and software is also possible.

As future work, we plan to provide a number of transformations at the process network level. These network transformations are for example, Channel Merging, in which a number of channels relating the same network processes are merged into a single one, Process Splitting, in which the loop structure of a process is unrolled resulting in this way a larger number of processes and Process Retiming, in which the iteration space of a process is rescheduled by applying unimodular transformations.

#### VI. ACKNOWLEDGMENTS

The authors of this paper would like to thank Alain Darte, Paul Feautrier and Dmitry Cheresiz for reading the article and providing valuable feedback.

#### REFERENCES

- [1] Uptal Banerjee. Dependence analysis. *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1997.
- [2] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In *Communicating Process Architectures - 2001, Proceedings*, pages 1–14, Bristol, UK, September 2001.

Algorithm name	Nb. Lines of code	Compilation time (M:S)	Nb. Processes	Nb. Channels	Channels type IOM-/IOM+/OOM-/OOM+
LU-Factor	30	00:40	5	26	14 / 6 / 5 / 1
QR-Decomp	26	00:13	6	12	12 / 0 / 0 / 0
SVD	65	05:34	8	69	35 / 4 / 30 / 0
Faddeev	35	00:27	9	23	19 / 3 / 1 / 0
Gauss-Elimin	26	00:14	4	11	7 / 0 / 1 / 3
DigBeamFormer	12	01:09	8	14	14 / 0 / 0 / 0
Motion Estim	78	01:57	11	93	93 / 0 / 0 / 0
M-JPEG	43	01:04	9	30	13 / 17 / 0 / 0

TABLE I  
EXPERIMENTAL RESULTS

- [3] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.
- [4] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19:179–194, July 1998.
- [5] Philippe Clauss and Vincent Loechner. Polylib. <http://icps.u-strabg.fr/Polylib>, February 2002.
- [6] Alain Darté, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.
- [7] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzter, P. Lieverse, and K.A. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.
- [8] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.
- [9] Eugène Ehrhart. *Polynômes arithmétiques et Méthode des Polyèdres en Combinatoire*. Birkhäuser Verlag, Basel, International series of numerical mathematics vol. 35 edition, 1977.
- [10] Paul Feautrier. Parametric Integer Programming. In *RAIRO Recherche Op?rationnelle*, 22(3):243-268, 1988.
- [11] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.
- [12] Tim Harriss, Richard Walke, Bart Kienhuis, and Ed Deprettere. Compilation from matlab to process networks realized in fpga. *Design automation of Embedded Systems*, 7(4), November 2002.
- [13] Peter Held. *Functional Design of Data-Flow Networks*. PhD thesis, Delft University of Technology, The Netherlands, 1996.
- [14] Vinod Kathail, Shail Aditya, Robert Schreiber, and Bob Rau. PICO: Automatically Designing Custom Computers. *IEEE Computer*, 35(9), September 2002.
- [15] Bart Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.
- [16] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000.
- [17] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.
- [18] Edward Lee et al. PtolemyII: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.
- [19] D.E. Maydan, S.P. Amarashinghe, and M.S. Lam. Data Dependence and Data-Flow Analysis of Arrays. In *Proc. of 5th Workshop of Languages and Compilers for Parallel Computing*, pages 434–448, August 1992.
- [20] Tom Parks. Bounded scheduling of process networks, December 1995. T. M. Parks, Bounded Scheduling of Process Networks, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California, Berkeley.
- [21] <http://www.picochip.com>.
- [22] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.
- [23] William Pugh and David Wonnacott. An Exact Method for Analysis of Value-based Array Data Dependences. In *Proceedings of the Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, Dec 93.
- [24] Edwin Rijpkema. *Modeling Task Level Parallelism in Piece-wise Regular Programs*. PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, September 2002.
- [25] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. "7th Int. Conf. Design, Automation and Test in Europe (DATE'04)"*, Paris, France, Feb 16 – 20 2004.
- [26] Paul Stravers and Jan Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *Proceedings of the Int. Symposium on VLSI Technology, Systems, and Applications*, April 2001. <http://www.synfora.com>.
- [27] Alexandru Turjan and Bart Kienhuis. Storage Management in Process Networks using the Lexicographically Maximal Preimage. In *Proceedings of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, June 24-26 2003.
- [28] Sven Verdoolaege, Kristof Beyis, Maurice Bruynooghe, Rachid Seghir, and Vincent Loechner. Analytical Computation of Ehrhart Polynomials and its Applications for Embedded Systems. In *2nd Workshop on Optimizations for DSP and Embedded Systems, (ODES'02)*, Palo Alto, CA, 2004.
- [29] Doran Wilde. A library for doing polyhedral operations. In *Technical Report PI 785, IRISA, Rennes, France*, 1993.
- [30] <http://www.xilinx.com>.
- [31] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, 2003.