SAMOS, II(1), 1–24 (2002)

The Compaan Tool Chain

Realizations of the Extended Linearization Model

Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere

Leiden Embedded Research Center, Leiden
University, Leiden, The Netherlands

At the Leiden Embedded Research Center, we are working towards a framework called Compaan that automates the transformation of digital signal processing (DSP) applications to Kahn Process Networks (KPNs). These applications are written in Matlab as parameterized nested loop programs. This transformation is interesting as KPNs are well suited for mapping onto parallel architectures. Although the KPN semantic always assumes that FIFO buffers can be used between processes, we have found cases in which the FIFO is not enough as data may arrive in the wrong order. To solve this order problem, we previously presented the Extended Linearization Model (ELM) that describes a mechanism to reorder tokens. The introduction of the ELM does not violate the Kahn Process Network semantics; we still use a FIFO between a Producer and Consumer. The ELM relays on some additional memory and a controller to perform the reordering. The ELM model can be implemented in different ways. In this chapter, we investigate four different realizations of the ELM. The realizations differ in the computational complexity of performing the reordering, the kind of reordering memory used, and the size of the reordering memory.

Copyright © 2000 by Marcel Dekker, Inc. www.dekker.com

I. Introduction

An appealing and fruitful methodology to deal with exploration or designing *applications - architectures pairs* has become known as the *Y-chart approach*, [1]. This approach embraces two fundamental notions: the *separation of concerns* and the *abstraction hierarchy*. The concerns are: the application, the architecture, and the mapping. The abstraction hierarchy, introduced in [2] as the *abstraction pyramid*, bridges - be it for exploration or synthesis purposes - the gap between high level application specification and low-level architecture specification by defining a number of abstraction levels and a corresponding *stack of Y-charts*. At each level, application models, architecture models, and mapping models must match to make exploration and synthesis feasible.

Several research groups around the globe are currently experimenting with this methodology, some explicitly and others implicitly. They are, naturally, all focusing on different application domains which lead to different views on this methodology. Applications in the realm of *automotive, multimedia,* and *communications* have different requirements, constraints, and boundary conditions which result in different challenges.

The Leiden Embedded Research Group focuses on applications that can be specified as parameterized affine Nested Loop Programs (NLPs). The group has been developing and implementing the *Compaan* tool chain to translate such applications from their imperative language specification into Kahn Process Networks (KPN) [3]. The application specification language is Matlab or C, and the tool-chain is a compiler through which a range of KPNs can be obtained for any given application specified as a parameterized NLP.

The processes in the Compaan generated KPNs are not (completely) specified in an imperative model of computation because the distance between that model and the models in which architecture components - in particular the processing units - are specified is too large. This is not specific to the application domain for which Compaan is an appropriate translation tool set; it is a problem that is revealed wherever the Y-chart methodology is used. Of course, the processes in KPNs may be specified in terms of more than one model of computation. For example, one could be obtained for the *Control Data Flow Graphs* model [4] or for one or more

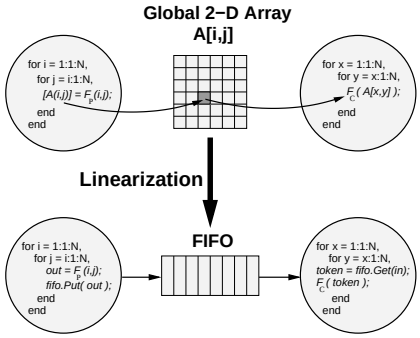

Figure 1. The Standard Linearization Model

Dataflow Network models [5].

The Process Network Model (PN) in Compaan is the *Kahn Process Network (KPN)* model [6], which consists of concurrent autonomous processes that communicate in a point to point fashion over unbounded FIFO channels using a blocking-read synchronization. The strength of a Process Network is that it uses no global memory and no global scheduler. This makes a KPN very appealing for further implementation into hardware [7].

In the Compaan KPN processes, each process executes an internal function following a local schedule. At each execution (also referred to as *iteration*) this function reads/writes data from/to different FIFOs. An input port domain (IPD) of a process is the union of the iterations at which the process's function reads data from the same FIFO. An output port domain (OPD) of a process is the union of the iterations at which the process's function writes data to the same FIFO. Each FIFO uniquely relates an input port to an output port forming to an instance of the classical *Producer/Consumer* pair [8].

One of the tools in the Compaan Tool Chain is *Panda*. Panda accepts as input the description of a Polyhedron Reduce Dependence Graph (PRDG) and transforms this PRDG into a Process Network (PN). This transformation is done in a number of steps. One of the steps involved is the *Linearization*, in which a high dimensional data structure (e.g., matrix $A[i,j]$) is linearized into a single linear stream of data. In case of Kahn process networks, the linearization model is a FIFO buffer as shown in Figure 1.

In the top part of this figure, a producer and consumer process are given that communicate the data array `A[i,j]` using global memory. In the linearization step, this communication is replaced with a FIFO buffer, leading to the producer/consumer processes given in the lower part of Figure 1. Observe that in the top part, the indices *i* and *j* are used to address matrix A. In the bottom part, the reference to A has been eliminated. The for-loops only describe an order and data produced by the function is placed on the FIFO buffer. There are cases, however, in which a FIFO as the Linearization Model (LM) no longer holds. If the order data is produced is different from the order data needs to be consumed, a FIFO buffer no longer is enough. In [9], we have proposed an extension to the LM, which we called the *Extended Linearization Model* (ELM). This model includes an additional reordering mechanism that consists of a *Controller* unit and some *Reordering Memory*. The ELM preserves the semantics of the KPN model. As we will show in this chapter, the ELM can be realized in different ways and each realization has its own strength and weakness. Based on these realizations, alternative hardware/software mappings of the Compaan generated network onto different platforms are feasible.

II. In Order/Out of Order case:

Consider the two KPN processes in Figure 2 with node domains `P` = { p, I } and `C` = {c, K}, respectively that are collections of atomic nodes `p`(i,j) and `c`(x,y) defined on the domains $I = \{(i,j) |\ 3 \leq j \leq N \wedge 1 \leq i \leq N-2\}$ and $K = \{\ (x,y) |\ 2 \leq x \leq N-1 \wedge 2 \leq y \leq N-1\ \}$, respectively. In the first process one of the OPDs is `O` = { `out`, J } that is a collection of atomic output ports `out`(i,j) defined on the domain $J = \{(i,j) |\ 3 \leq j \leq N \wedge i+2 \leq j \wedge 1 \leq i \leq N-2\}$. In the second process one of the IPDs is `I` = { `in`, L } that is a collection of atomic input ports `in`(x,y) defined on the domain $L = \{(x,y) |\ 2 \leq x \leq N \wedge x \leq y \wedge 2 \leq y \leq N\}$. There is a mapping M $(i = x-1; j = y+1;)$ relating these two port domains. Hence, these two ports form a Producer/Consumer pair. A token produced by the atomic node `p`(i,j) is put on the FIFO channel reserved for the edge domain (`O`, `I`) through the atomic output port `out`(i,j), and will be consumed by the atomic node `c`(x,y) through the atomic input

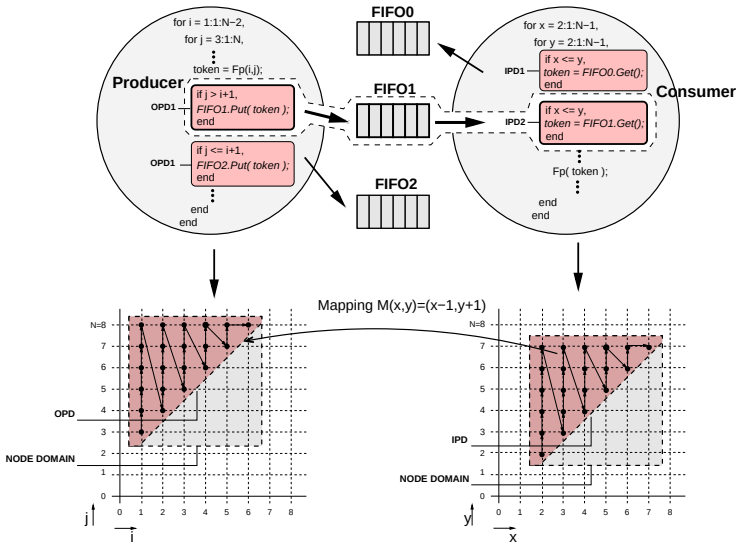

Figure 2. A Producer and Consumer process. Of the Producer we show the output port domains (OPDs) and of the Consumer, we show the input port domains (IPDs). Each OPD is uniquely connected to another IPD via a FIFO. Over this FIFO, tokens are communicated that adhere to the mapping given by the mapping matrix M. In this example, OPD1 is connected to IPD2 via FIFO1. The Producer/Consumer with the FIFO form an instance of the classical consumer/producer pair.

port `in`(i+1, j-1) that gets the token from this channel.

Since the KPN processes are sequential processes, no two atomic ports in a port domain are active at the same time. That is, there is an order among the atomic output ports in an output port domain, and there is an order among the atomic input ports in the corresponding input port domain. In [9], we have defined the *rank* function that expresses in a pseudo-polynomial form this order of execution in a particular domain. The rank function is derived using the Ehrhart theory that expresses the number of integral points inside of a polytope as a pseudo-polynomial expression [10]. A pseudo-polynomial is a polynomial with periodic coefficients. This theory has been extended recently for parameterized polytopes [11].

As a consequence, the expression of the rank function is in general a set of pseudo-polynomial expressions depending on the parameters. Examples of the rank functions will be shown later when various realizations of the ELM are discussed.

In Compaan, the sequential ordering of atomic nodes firing in a node domain is in *lexicographical order*, which means these nodes are scheduled according to a loop nest. The ordering in which tokens are put on a channel is the same as the order in which atomic nodes are fired. Because the channel is a FIFO channel, a consumer can only get the tokens from the channel in the same order. This represents the *in-order* case. However, depending on the lexicographical schedule of the consumer's atomic nodes, the consumption of the channel tokens may follow a different order than the order in which these tokens were put on the channel. This represents the *out-of-order* case. To work correctly in the out-of-order case, a Consumer needs a mechanism to restore the consumption order. This mechanism relays on the use of private *reordering memory* for temporary storage of tokens. Once stored, the tokens can be consumed in the correct order. This reorder mechanism is modeled as the ELM.

III. The Extended Linearization Model in more detail

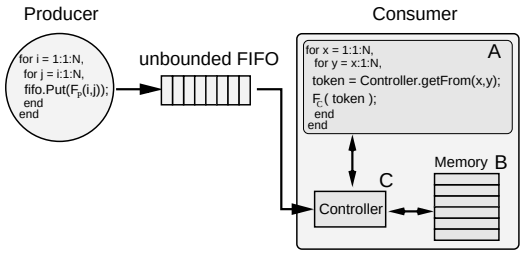

Figure 3. The Extended Linearization Model

The main elements in the Extended Linearization Model are the local reordering memory and the Controller. Because the tokens can no longer be read directly from the FIFO, as they may arrive in the wrong order, they are delivered by the Controller to the function unit. In this way, the

Controller takes care of supplying tokens to the consumer Function in the right order. In Figure 3, a schematic representation is given of the ELM. It shows the Consumer process (A), the Reorder Memory (B), and the Controller (C).

A. The Process Description (A)

The process description in the ELM is different from the process description when using the LM. Instead of getting tokens directly from a FIFO, the function gets its tokens from the Controller. Hence function call *fifo.Get* (See the lower part of Figure 1) is replaced with the call to the Controller function *getFrom*.

B. The Memory (B)

The Memory stores tokens allowing the Controller to reorder tokens into the order required by the Consumer process. Two kinds of memory are possible: Random Access Memory (RAM) and Content Addressable Memory (CAM). The two kinds of memory differ in the way they are addressed. The implementation of the Controller depends on the type of the memory.

C. The Controller (C)

The Controller converts the sequence tokens are produced into the sequence they have to be consumed. The Controller performs this reordering by addressing the reordering memory (B). This functionality is exposed externally to the Consumer process by the function *getFrom(x,y)* that returns the token to function F_C for an arbitrary iteration point (x, y).

The behavior of the Controller is shown in pseudo code in Figure 4. The `getReadAddress(x,y)` determines the memory address of the token needed at the iteration (x, y). Next, the Controller checks whether the token is already available at that address by calling the function `emptyMem`. If the token is present, it is read from that address by calling the

```
Token t getFrom(x,y) {
   double address = getReadAddress(x,y);
   if( !emptyMem(address) ) {
     return readFromMem(address);
   } else {
     return readFromFifo(address);
   }
}
```

Figure 4. The components in the Controller.

function `readFromMem`. Otherwise, the Controller starts to read tokens from the FIFO and stores them in the memory until the desired token arrives at the address of interest. The procedure of reading from FIFO is initiated using the function call `readFromFifo`. Storing tokens into the memory implies that for each token read from the FIFO, a certain address is generated. Depending on the type of memory used, different procedures are available to generate this address. These procedures are realized as the function *getWriteAddress* inside the function `readFromFifo` .

IV. Realizations of the Extended Linearization Model

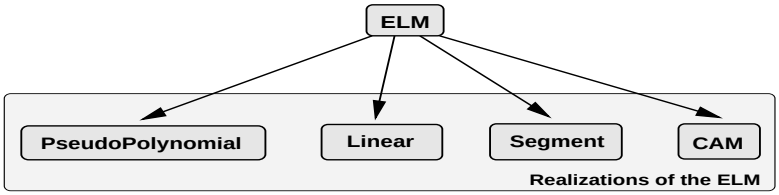

Figure 5. Four Model Instances

The ELM can be realized in four different ways as shown in Figure 5. The realizations differ by the way the function *getReadAddress* and function *getWriteAddress* are implemented and by the type of memory used as reordering memory. To compare the four different realizations, the follow-

ing three characteristics are relevant:

The complexity of the addressing mechanism the computational complexity of the controller functions *getReadAddress* and *getWriteAddress*.

The dimension of the reordering memory the number of the storage locations needed to perform the reordering.

The generality of the realization the class of algorithms for which Compaan can derive KPNs.

To introduce the four realization, we use as an example the Producer/Consumer pair given in Figure 6. The graphical representation of the domain descriptions of the Producer/Consumer pair is shown in the top part of Figure 7. Because the order the Producer produces data is different from the order the Consumer consumes, an ELM realization is needed in the linearization of the Producer/Consumer pair.

```
for (int i=1;i<=N+2;i++){
   for (int j=1;j<= N;i++){
      if (2*j >= i+6){
          a[i,j] = Fp();
      }
   }
}
```

Producer

```
for (int y=4;y<= N;y++){
   for (int x=1;x<= N+2;x++){
      if (x <= 2*y-6){
          Fc(a[x,y]);
      }
   }
}
```

Consumer

Figure 6. Running Example

V. PseudoPolynomial realization

The PseudoPolynomial realization is based on the fact that the order of the iterations inside an OPD can be expressed as a pseudo-polynomial, which is the rank function discussed earlier in this chapter. In general, the *getReadAddress* function of the Controller is a pseudo-polynomial function. In Figure 7, the iteration points of the OPD are perfectly enclosed by a shape that we call the *linearization shape*. The pseudo-polynomial expression is computed by calculating the $rank$ function inside the Linearization shape. This consists of adding several pseudo-polynomials R_1,

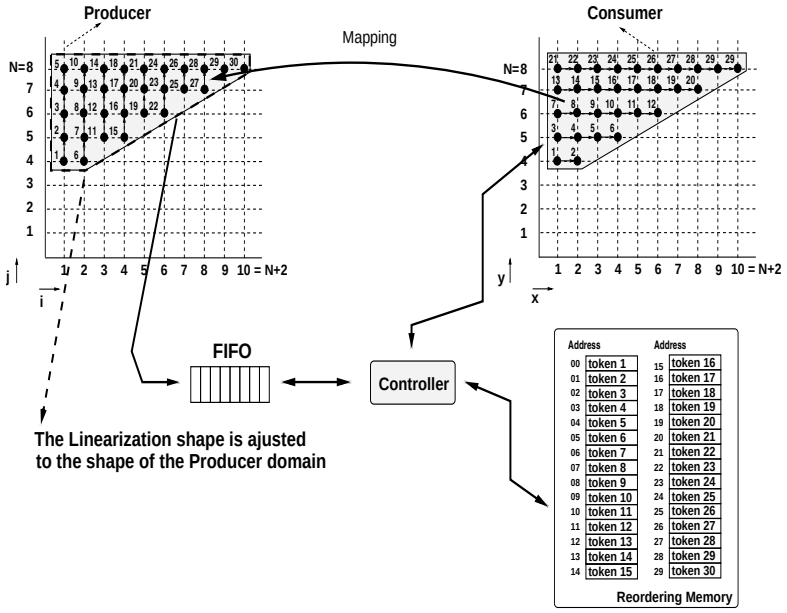

Figure 7. The PseudoPolynomial realization.

P_2, ... P_n, where n is equal to the dimension of the Linearization shape. For our running example, the rank is the sum of the two pseudo-polynomials P_1 and P_2:

$$\begin{aligned} rank(i,j) &= P_1(i,j) + P_2(i,j) \\ P_1(i,j) &= (i-1) * N + (-1/4) * i^2 - 2 * i + [2,9/4]_i \\ P_2(i,j) &= j + (-1/2) * i + [-3,-7/2]_i \\ rank(i,j) &= -1/4 * i^2 + (N - 5/2) * i + j + [-1,-5/4]_i - N. \end{aligned} \tag{1}$$

To obtain the *getReadAddress* function, the rank needs to be composed with the mapping $M(x,y)$ and the result is equal to :

$$\begin{aligned} getReadAddress(x,y) &= rank(i,j) \circ M(x,y) \\ &= -1/4 * x^2 + (N - 5/2) * x + y \\ &\quad - [1,5/4]_x - N. \end{aligned} \tag{2}$$

The $rank$ polynomial contains all the information needed by the Consumer process to reorder the token correctly. For this realization, tokens are written into the reorder memory following the sequence into which they arrive from the FIFO. Therefore, the function *getWriteAddress* is a simple increment. The dimension of the reordering memory is equal to the number of iteration points in the OPD. For the Producer/Consumer of Figure 6, the dimension is equal to $(N-4)*(N+2)/2$. The computational complexity of addressing the reordering memory can be quite large. It requires the evaluation of a pseudo-polynomial expression like, for example, the one given in equation 2. In general, the PseudoPolynomial realization is valid only for the cases when an OPD is a polytope. Under certain conditions, the realization can be extended for cases an OPD is not a polytope [12].

VI. Linear realization

The Linear realization is based on the classical Linearization of an n-dimensional array into a one-dimensional array [13, 14]. The classical Linearization shows that a rectangular shape can be addressed using a simple polynomial. Inspired by this concept, we relax the Linearization shape to the smallest rectangular that includes the producer domain (OPD). Consequently, the *getReadAddress* that results is always a simple linear function. The rectangular Linearization shape is shown in Figure 8, and the $rank$ function is as follows:

$$rank(i,j) = (N-3)*(i-1)+j-4. \tag{3}$$

The *getReadAddress* function is obtained by composing the rank function with the mapping function $M(x,y)$, and the final polynomial expression is :

$$\begin{aligned} getReadAddress(x,y) &= rank_P(i,j) \circ M(x,y) \\ &= (N-3)*(x-1)+y-4. \end{aligned} \tag{4}$$

The consecutive order inside the Linearization shape, however, can get disturbed. This happens when an OPD doesn't have a rectangular shape and therefore, more iteration points are enclosed by the Linearization shape than necessary. As a consequence, these additional iteration points are also

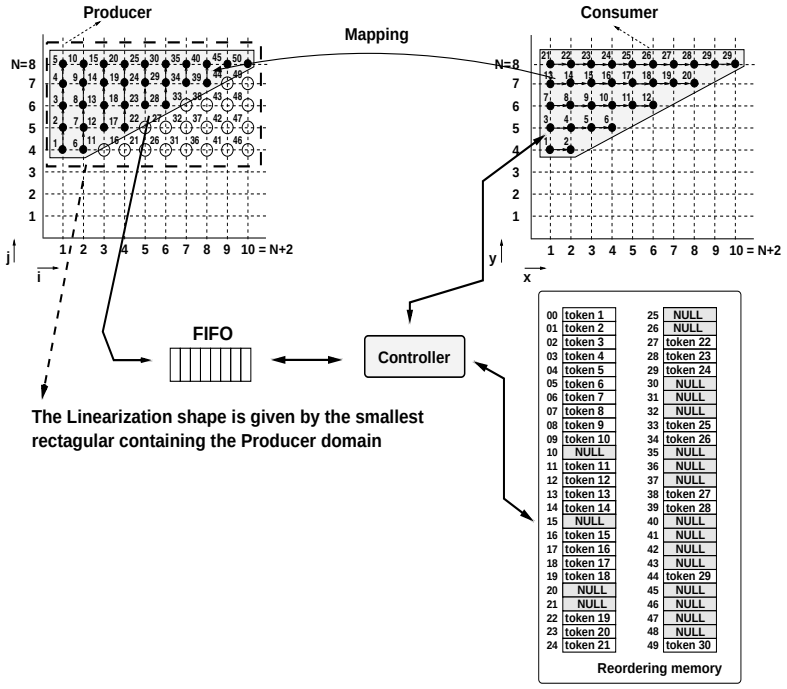

Figure 8. The Linear realization

ranked by the $rank$ function, disturbing the consecutive order. Looking at Figure 8, we see that after iteration 10 follows iteration 11. However, iteration 11 does not belong to the OPD. The next iteration belonging to the OPD has rank 12 and hence the order becomes 10, 11, 12. Consequently, the Controller cannot rely any longer exclusively on the order tokens are read from the FIFO; the eleventh token read from the FIFO should be written at the address 12. Therefore, the Controller cannot use a simple increment for the *getWriteAddress* function.

To re-create the correct sequence of addresses, the Controller relays on a function that assigns to incoming tokens the correct order number inside the OPD. This function is called the *recover* function. This function reimplements at the Consumer side the logic used to schedule the iteration points inside the OPD.

The advantage of this realization, is that the function used to address the

reordering memory is always a linear expression depending on the coordinates of the consumer iteration point. A disadvantage is that need for the *recover* function. Moreover, the extra iteration points enclosed by the linearization shape result in empty memory slots (represented by the "Nulls" in the memory in Figure 8). In the example, the memory requirement is equal to the dimension of the Linearization shape, i.e., to $(N-3)*(N+2)$, but only half of this space is actually used.

VII. Segment realization

The PseudoPolynomial realization results in good memory usage, but the addressing formula can be very complex because of the irregularities it contains as expressed by the periodic coefficients. On the other hand, the Linear realization results in simple addressing but potentially wastes a lot of memory. We now present the Segment realization that combine the best features of the two approaches discussed so far: simple addressing mechanism and efficient memory usage.

The Segment realization is based on the fact that pseudo-polynomials can be decomposed into a *linear part* and a *non-linear part* as shown in Figure 10. The linear part describes the consecutive order, the non-linear part described the non consecutive order. At the Producer side, the order changes at iteration points at which the innermost nested loops start to iterate again from their lower bound value. We say that a *non-linearity* occurred at iteration point (IP) and using the notion of these IPs, the Segment realization computes the value of the pseudo-polynomial using a *Segment Number* and a *Segment Displacement* as is shown in Figure 10. How the Segment Number and Displacement are computed is explain later on in the chapter. Because the segment number and displacement are pre-computed, a pseudo polynomial cannot be evaluated in a parameterized way, as is possible in the PseudoPolynomial realization. Hence, parameter values in a NLP have to be fixed in order to use the Segment realization.

Writing data into the reordering memory occurs in the same way as in the PseudoPolynomial realization. The Controller writes tokens in the *Reordering Memory* as they arrive from the FIFO. The detection of the IPs at which the consecutive order get disturbed, is done by the *recover*

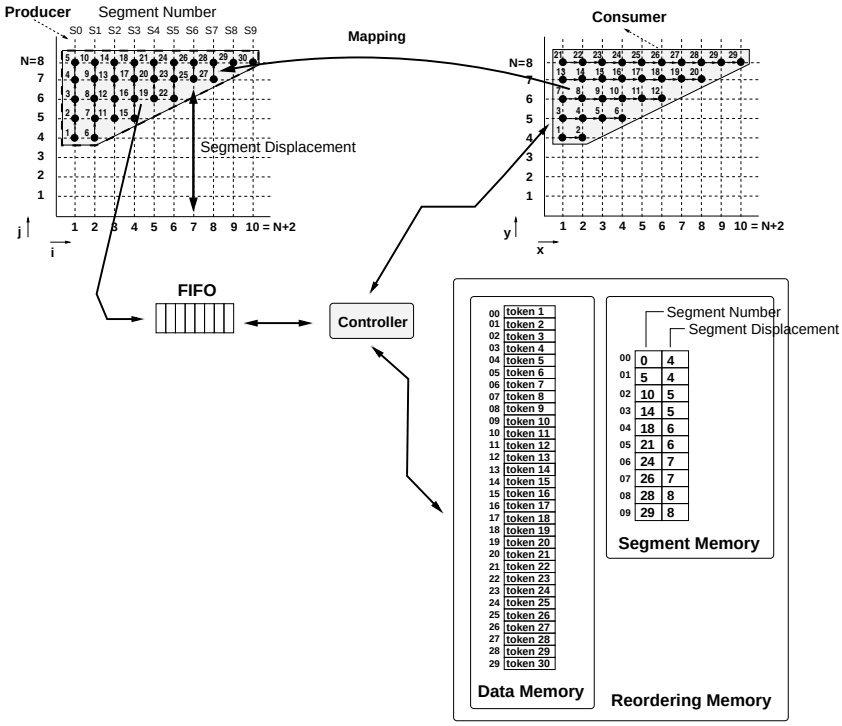

Figure 9. The Segment realization

function that duplicates the Producer for-loops at the Consumer side. With each occurrence of an IP, a *Segment Number* and *Segment Displacement* is associated. Each such number pair is used by the Controller to determine the value of the Pseudo polynomial. Let's see how writing and reading takes place in the Segment realization. The writing is implemented in the *getWriteAddress* and the reading is implemented in the *getReadAddress*

Writing a token happens in the following way. Initially, the Controller contains an internal counter that is set to zero. The $recover$ function keeps track of whether the order is linear or non-linear. if the order is linear, a token is read from the FIFO, and the internal counter is incremented by one. If the order is non-linear, the Controller allocates a new entry in the *Segment Memory*. It writes in the entry the current value of the iterator in the *Segment Displacement* field and the currently value of the counter in

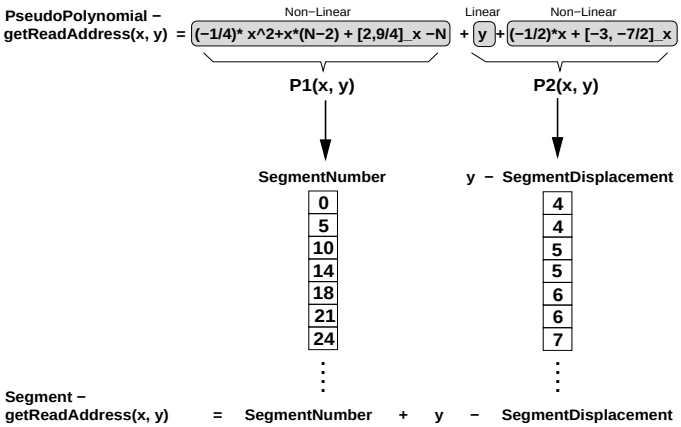

Figure 10. Computation of a Pseudo Polynomial using a Segment Number and a Segment displacement stored in the Segment Memory.

the *Segment Number* field.

In Figure 9, the Producer starts at iteration $(1, 4)$, which immediately results in an IP for iterator i. Consequently, an entry is allocated in the segment memory at address 0. The counter has a value of 0 and the iterator is equal to 4, leading to entry (0,4) at address 0. Next, iterator i moves consecutive to iteration $(1, 8)$. At the next iteration of i, an IP occurs again. A new entry is generated at address 1. The counter value is equal to 5 and the value of i is again 4, leading to the (5,4) entry at address 1 and so one.

For a particular iteration point of the Consumer, the Controller determines the address from where data has to be consumed using a three-step procedure. The three steps are:

$$\begin{aligned} &\text{step1:} \quad (i, j) = Map \circ (x, y). \\ &\text{step2:} \quad Segment = SegmentMemory(i - i_{start}). \\ &\text{step3:} \quad address = Segment_{Number} + j - Segment_{Displacement}. \end{aligned} \tag{5}$$

In Figure 9, the Producer starts from the iteration (i_{start}, j_{start}), which is equal to $(1, 4)$. Suppose the Consumer wants to obtain the token for iteration $(4, 6)$. In step 1, the iteration is mapped in an iteration at the Producer and is equal to $(4, 6)$ (i.e., the mapping is identity). In step 2,

the segment is found associated with this iteration. In step 2, i_{start} is equal to 1 and i is equal to 4. Hence, the segment number is 3, which is address 3 in the segment memory were entry $(14, 5)$ is stored. In step 3, the address in the reordering memory is calculated as 14 + 6 - 5 = 15. At address 15, the token is stored that was generated at the 16th iteration by the Producer. This is the token needed by iteration $(4, 6)$ of the Consumer as can be verified by inspection in Figure 9. The memory size of this realization is equal with the size of the DataMemory plus the size of the SegmentMemory. In our example the size of the DataMemory is $N^2/2 + 1/2N - 6$ (the same as the memory size from the PseudoPolynomial realization) and the size of the SegmentMemory is $N + 2$. Hence, the total memory size is equal to $N^2/2 + 3/2N - 4$.

VIII. CAM realization

The CAM realization uses a Content Addressable Memory (CAM) as reordering memory. In a CAM, a *key* is used instead of an address to access the content of the memory. The entry used in the CAM realization is given in Figure 11. It shows that each entry in the CAM consists of a key, the token associated with the key, and a field called *multiplicity*. We explain later what the term multiplicity means. The CAM approach works, because to each token produced at the producer OPD an *unique* key can be associated. The Controller can reproduce this key to obtain the token the Consumer requires at a particular iteration.

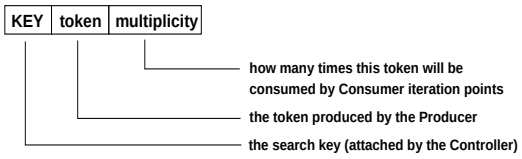

Figure 11. The CAM entry

For the CAM realization, the function *getReadAddress* generates a key instead of an address. The generation of the unique key can be done in different ways. We compute the $rank$ function inside the Producer based on

an Node domain instead of an OPD. Another possibility would have been to use a classical linearization polynomial. In general, the shape of a Node domain results in a simple polynomial instead of a pseudo-polynomial and it therefore easily calculated. Using the $rank$, a unique number is associated that is equal to the order of the iteration inside the Node domain of the Producer. To illustrate this, consider again the Producer/Consumer pair from Figure 6. Suppose that the Node domain is defined as

$$C = \{(i,j) |\ 1 \leq i \leq N \wedge 1 \leq j \leq N+2\},$$

Then the $rank$ is given by

$$rank(i,j) = i * N + j.$$

By composing the $rank$ with the mapping $M(x,y)$ we obtain the *getReadAddress* function:

$$getReadAddress(x,y) = x * N + y. \tag{6}$$

For a given iteration point (x,y) of the Consumer process, the *getReadAddress* function determines the unique key for that iteration using equation 6. For this key, the Controller checks (using function *EmptyMem*) if a token already exists in the CAM by searching all keys for a match. If no match can be found, the token is not stored yet.

If the key exists, the token associated with the key is retrieved from the CAM by function *readFromMem*. If the token doesn't exist, the Controller keeps loading data from the FIFO into the CAM. This happens in function *ReadFromFifo*. To each token the Controller loads, it attaches an unique key given by the function *getWriteAddress* and multiplicity number. Loading data from FIFO stops upon arrival of the token for which the key (as given by *getWriteAddress*) is the same as the key the Controller is searching for (as given by *getReadAddress*). The function *getWriteAddress* is based on a $recover$ function similarly to the recover function from the Linear realization.

In general, a token is read only once by the Consumer process. There are cases in which the same token is read more than once by the Consumer process. This called a *broadcast*. A read from a FIFO is destructive and in case of a broadcast, this would mean that a token needs to be send over the FIFO as many times as needed, or that a token needs to be stored in

memory and read from memory as many times as needed. In the CAM realization, we implemented the latter option as it is more efficient.

To keep track of how many times a token is to be read, we have introduce the notion of *multiplicity* [12], which indicates how many times a particular token needs to be read by the Consumer process. Each time a token is consumed, its multiplicity is decremented. When the multiplicity reaches zero, no other iteration will need that token and it can be erased. That location can be reused by other tokens. Hence, using multiplicity, the Controller is able to free memory locations and consequently, this realization uses the smallest possible amount of memory. The memory size (MS) of the CAM is given by the next formula:

$$\text{MS} = \max_{(x,y)\in C}(read(x,y) - rank_{Consumer}(x,y)) + 1. \qquad (7)$$

where C represents a sub-domain of the Consumer domain where no two points read the same token. In the case from Figure 6, C is the whole Consumer domain. The $read$ function is the same as the *getReadAddress* function from the PseudoPolynomial realization:

$$read(x,y) = -1/4 * x^2 + (N - 5/2) * x + y - [1, 5/4]_x - N. \qquad (8)$$

and the $rank_{Consumer}$ is the function that gives the order of the Consumer iteration points:

$$rank_{Consumer}(x,y) = y^2 - 7y + x + 11. \qquad (9)$$

According to equation 8 and 9 it resultes that:

$$\text{MS} = \max_{(x,y)\in C}(-1/4 * x^2 + (N - 7/2)x - y^2 + 8y - N - [11, 45/4]_x).$$

The maximum of this formula inside the C domain can be derived using analitycal methods. In our case for $N = 8$, we have $MS = 10$. For more informations about the read and the rank function, we refer to [9]. The key to efficient memory usage is the ability to compute the multiplicity for a token. However, this multiplicity is again computed using the Ehrhart theory and may again be a pseudo-polynomial.

IX. Comparing the different realizations

In the previous sections, we have shown four different realization for the ELM. Each realization has its strength in terms of efficiency of memory usage and computational complexity of address the memory. In this section we make some general remarks about the different realizations and summarize the strengths and weaknesses of the four realizations.

A. General Remarks

i. Linearization Shapes

In the realizations, the Linearization shape of the OPD determines the complexity of the *getReadAddress* implementation. We indicated that when rectangles are enforced, simple polynomials result. There are application domains where the rectangular shape is the natural Linearization shape, for example, in imaging. In those cases, the Linear realization doesn't have the disadvantage of memory wastes and the need for a $recover$ function. On the other hand, we found more complex Linearization shapes in advanced signal processing algorithms. In algorithms like QR or SVD, triangular shapes are typical leading to complex pseudo polynomials.

ii. Parameterized versus Static realizations

We solved the Linearization under the assumption that we want to keep the problem parameterized in the original parameters of the loop-bounds of the parameterized NLPs. If, however, we need to provide a realization for specific values of these parameters, we can come up with a much simpler realizations of the Controller. The Segment realization already shows that. In general, if we can evaluated the *getReadAddress* a priori, the Controller becomes a simple look-up table.

iii. RAM versus CAM

RAM is the most commonly used form of memory. It is simple, cheap and widely available on todays FPGAs. But more and more, CAMs are also becoming available. Now a days, there are FPGA platforms available on the market that supports CAM blocks with high speed search time [15].

iv. Dense Polytopes

In the examples shown so far, we assumed that all nodes in the OPD and IPD can be enclosed by a Linearization shape. There are cases, however, in which we can find the exact shape, but still not all points are part of the enclosure. We refer to these points are *holes* and they are introduced when a for-loop is used with a stride other than one or when linear expressions are used that contain operators like mod, div, floor, ceil, max, or min. The holes affect the generality of the realizations presented in this chapter. Not all of the discussed realizations can handle holes. For example, if a Linearization shape encloses holes, these holes get also ranked, thereby disturbing the consecutive order.

v. Recover Function

In three of the presented realizations, the function *getWriteAddress* is based on the *recover* function. For each token read from the FIFO, this function recovers the iteration at which this token was produced inside the IPD. Basically such function duplicates the control from the IPD as a finite state machine, which can be computationally expensive.

Instead of using the *recover* function at the Consumer, another approach would be to tag the tokens produced at the OPD with additional information. In this way, the Controller and memory have the same function as the *matching unit* found in classical Dataflow architectures [16]. The problem in these matching units was to find a lower bound on its memory, such that a program wouldn't deadlock. We have shown in equation 7 that we can determine a lower bound on the memory such that no dead-lock occurs given the class of parameterized NLPs.

vi. Practical limitations

The Pseudo-Polynomial realization depends very much on the ability to calculate the $rank$ functions. We rely on the Polylib library [17], to compute the rank function. Although this library has proven to be quite stable and useful, this implementation of the Ehrhart theory is not always able to compute the $rank$ function. By selecting the Linear, the Segment, or the CAM realization, we are always able to come up with a representation of a KPN.

B. Summary

We have presented four different realization for the ELM. In Figure 12, we compare these realizations for the Producer/Consumer given in Figure 6.

Linearization Model	Memory size	N=8	Computational Complexity	recover	Generality
Pseudo	$N^2/2 + 1/2 \cdot N - 6$	30	$C1$	No	No
Linear	$N^2 - N - 6$	50	$C2 \ll C1$	Yes	Yes
Segment	$N^2/2 + 3/2 \cdot N - 4$	40	$C3 \sim C2$	Yes	Yes
CAM	$max(read - rank)$	10	$C4 \sim C2$	Yes	Yes

Figure 12. Comparision of the ELM realizations

The table shows the memory requirements in a symbolic way for parameter N and when $N = 8$, the computational complexity of addressing the memory (as done by function *getReadAddress*), whether a *recover* function is needed, and finally the generality of the approach. We can see that the Segment realization uses more memory than the PseudoPolynomial realization because the segment part consumes some memory. The advantage of the Segment realization is that the Controller can fill the memory in the same order tokens arrive. The Segment realization uses less memory than the Linear realization. If you look to the computational complexity of addressing in the Segment or Pseudo-Polynomial cases, you can see that the complexity is less for the Segment realization although in both cases a pseudo polynomial is evaluated. Finally, we observe that the CAM realiza-

tion uses the least amount of memory but may require a relatively complex addressing mechanism since the CAM realization requires the computation of unique key.

X. Conclusions

In this chapter, we have presented the Extended Linearization Model (ELM). This model was introduced to solve out-of-order communications of tokens between a Producer and a Consumer process. The ELM adds to a process some additional memory and a controller without violating the Kahn Process Network semantics; we still use a FIFO between a Producer and Consumer. The Controller uses the local memory to re-order the tokens in the order the Consumer expects the tokens. In the realization of the ELM, the implementation of the Controller is the difficult part.

To implement the Controller, we make a lot of use of the $rank$ function. This function assigns to an arbitrary iteration a unique rank number that indicates when it is produced. The $rank$ function is in general a pseudo-polynomial. We exploit the ability to derive such polynomial at compile time to find realizations for the ELM at compile time. In the realizations, we assumed that we only exchange tokens over a FIFO without any additional information. We did not assume tagging of tokens.

When realizing the ELM, we have seen that four different realizations exist. The first realization is the pseudo-polynomial realization. It uses exclusively pseudo-polynomials to solve the reordering case. The advantage is that we can solve the reordering in a parameterized way. Because in the most general case pseudo polynomial are involved, the implementation can be computationally complex. Also, the pseudo-polynomial can in practice not always be calculated. If we relax the Linearization shape to the smallest rectangular that encloses all iterations, we obtain a more simple implementation. However, this might be at the expense of inefficient memory usage. If we want to avoid complex addressing and efficient memory usage, the Segment realization is a good choice, although the solution is not longer parameterized. Finally, we showed that we can use a key instead of an address to retrieve the proper tokens. The calculation of this key is in general a simple polynomial which is easy to realize. Also, the CAM

realization requires the least amount of memory to solve the reordering of tokens.

The KNPs derived by Compaan can be simulated using the YAPI framework [18] or using the PN-domain in Ptolemy II [19]. In both cases, we have to implement the presented realizations in software. We are currently able to implement in software, at compile time, the PseudoPolynomial, and CAM realization. For these realizations, we have shown that we can derive correct implementations. We verified this by running Compaan on a set of applications written as parameterized NLPs.

References

1. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97), Zurich, Switzerland (1997) 338 – 349
2. Kienhuis, B., Deprettere, E., van der Wolf, P., Vissers, K. In: A Methodology to Design Programmable Embedded Systems. Volume 2268 of LNCS. Springer Verlag (2002) 18 – 37
3. Kienhuis, B., Rijpkema, E., Deprettere, E.F.: Compaan: Deriving process networks from matlab for embedded signal processing architectures. In: 8th International Workshop on Hardware/Software Codesign (CODES'2000), San Diego, USA (2000)
4. Wolf, W.: Computers as Components - Principles of Embedded Computing System Design. Morgan Kaufmann Publishers, Inc. (2001)
5. Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83** (1995) 773–799
6. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74, North-Holland Publishing Co. (1974)
7. Harriss, T., Walke, R., Kienhuis, B., Depettere, E.: Compilation from matlab to process networks realized in fpga. Design Automation of Embedded Systems **7** (2002)
8. Ben-Ari, M.: Principles of Concurrent Programming. Prentice Hall

(1982)
9. Turjan, A., Kienhuis, B., Deprettere, E.: A compile time based approach for solving out-of-order communication in kahn process networks. In: Proceedings of the IEEE 13th International Conference on Application-specific Systems, Architectures and Processors (ASAP'02), San Jose, California (2002)
10. Ehrhart, E.: Polynômes arithmétiques et Méthode des Polyédres en Combinatoire. International series of numerical mathematics vol. 35 edn. Birkhäuser Verlag, Basel (1977)
11. Clauss, P., Loechner, V.: Parametric analysis of polyhedral iteration spaces. Journal of VLSI Signal Processing **19** (1998) 179–194
12. Rijpkema, E.: Modeling Task Level Parallelism in Piece-wise Regular Programs. PhD thesis, University Leiden, LIACS, Leiden, The Netherlands (2002)
13. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986) ISBN 0-201-10088-6.
14. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Inc. (1997)
15. Xilinx: Memory Application Notes for Virtex-II Devices. www.xilinx.com (2001) pag 505-506.
16. Veen, A.H.: Dataflow machine architecture. ACM Computing Surveys **18** (1986) 366–396
17. Clauss, P., Loechner, V.: Polylib. http://icps.u-strabg.fr/Polylib (2002)
18. de Kock, E., Essink, G., Smits, W., van der Wolf, P., Brunel, J., Kruijtzer, W., Lieverse, P., Vissers, K.: Yapi: Application modeling for signal processing systems. In: 37th Design Automation Conference, Los Angeles, CA (2000)
19. Davis II, J., Hylands, C., Kienhuis, B., Lee, E.A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Tsay, J., Vogel, B., Xiong, Y.: Ptolemy ii - heterogeneous concurrent modeling and design in java (2000)