Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 235–258

Systolic multiplication – comparing two automatic systolic array design methods

Laura Ruff
Babeş-Bolyai University, Cluj-Napoca
Faculty of Mathematics and Computer Science
email: laura@cs.ubbcluj.ro

Abstract. This paper provides a comparison between two automatic systolic array design methods: the so called *space-time transformation* methodology (a unifying approach to the design of VLSI algorithms [14], [15]), and a *functional–based design method* (see [6], [9], [10]).

The advantages (and possible disadvantages) of each method are pointed out by representative case studies (variants of systolic arrays generated with both design methods).

Many algorithms were already parallelised using the efficient technique of space-time transformations. However, it also has some drawbacks. It may be hard to formulate the problem to be solved in the form of a *system of uniform recurrence equations*, which is the usual starting point for this method. On the other hand, the space-time transformation method depends heavily on finding an affine timing function, which can also lead to complex computations.

The functional-based method exploits the similarity between the inductive structure of a systolic array and the inductive decomposition of the argument by a functional program. Although it is less general in the sense that it generates systolic arrays with certain properties, its most significant advantage is that it needs to investigate the behaviour of only the first processor of the systolic array, while other methods (as the space-time transformation method, too) must work with an array of processors. Moreover, the method is based on rewriting of terms (according to certain equations, which are general for function definitions

AMS 2000 subject classifications: 68M07, 65Y05, 68W35
CR Categories and Descriptors: B.6.3 [Design Aids]: Subtopic - Automatic synthesis B.6.1 [Design Styles]: Subtopic - Parallel circuits
Key words and phrases: systolic array, automatic systolic array design, space-time transformations, functional approach

and systolic arrays), thus the resulting systolic algorithm is certified to be correct, and the method itself is relatively easy to automatize.

1 Introduction

The first examples of systolic arrays (very efficient special purpose parallel computing devices) and the systolic algorithms running on them were conceived in an ad–hoc manner, requiring a great amount of intuition and creativity from their inventors.

Later several [semi]automatic systolic array design methods were proposed (a short survey can be found in [16]). Most of these systematic methods use an *iterative view* of systolic arrays: the arrays (and the computations) are represented as multidimensional matrices of a certain size (in fact some methods only work for a fixed size, the problem cannot be parametrized). This kind of representation leads to complex operations over the multidimensional index space, on the other hand, due to the symmetric organisation of systolic structures, there are many repetitions in the design process.

The most widespread and also the most general method is referred to as the *space-time transformation method*. This is in fact a unifying approach to the design of systolic arrays, which incorporates the main ideas used in several automatic synthesis methods. The work of many researchers like Quinton, Robert, Van Dongen [12, 14, 13], Delosme and Ipsen [1], Nelis and Deprettere [8] relies on it. A review of the main ideas involved in the space-time transformation method is presented by Song in [15].

Many algorithms were already parallelised with this efficient method, however it also has its drawbacks.

The problem to be solved should be formulated as a *uniform recurrence equation system*, which is sometimes not an easy task. The uniformisation of linear recurrence equations was tackled by Quinton and Dongen [13], Fortes and Moldovan [3] and others but it is still not definitely solved.

The space-time transformation method heavily depends on finding an adequate affine timing function. The problem with finding such a function is that one needs to solve a linear equation system, which is usually a tedious and difficult task, on the other hand it is only possible for systems having certain properties.

As we already mentioned, the most design methods, thus the space-time transformation method, too, uses an *iterative* approach to the problem. In contrast, our design method presented in [6] follows a *functional view*: a linear systolic array is composed of a *head processor* and an identical *tail array*. Similarly, functional programs for list operations describe how to compute the head and the tail of the result in function of the head and the tail of the arguments.

Our design method exploits this similarity, thus the synthesis problem can be solved by [essentially] rewriting of the functional programs.

In this paper we compare this functional-based method with the space-time transformation method using some representative case studies. Our purpose is not the detailed presentation of the two methods (one can find such descriptions in [14, 15] –about the space-time transformation method–, respectively in [6, 9, 10] –about the functional/based method–). However, we would like to point out the advantages (or disadvantages) of the two distinct methods through some practical examples.

2 Systolic array design for polynomial multiplication

We start with a simple problem, the polynomial multiplication.
Let A and B two univariate polynomials of degree $n-1$ and $m-1$, respectively:

$$\begin{array}{l} A = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_{n-1} * x^{n-1} \\ B = b_0 + b_1 * x + b_2 * x^2 + \ldots + b_{m-1} * x^{m-1} \end{array}$$

We denote the product of A and B with C (polynomial of degree $n + m - 2$):

$$C = A * B = c_0 + c_1 * x + c_2 * x^2 + \ldots + c_{m+n-2} * x^{n+m-2},$$

where

$$c_k = \sum_{i+j=k} a_i * b_j, \quad \forall k, \ 0 \le k \le m+n-2; 0 \le i \le n-1, 0 \le j \le m-1 \tag{1}$$

2.1 Solutions to the problem using the space-time transformation method

In order to be able to apply the space-time transformation methodology to the problem, the coefficients of C should be given in a recursive way, thus (1) is not an adequate formulation to start up with. We need a *uniform recurrence equation system* which is a subclass of linear recurrence equation systems. Only such systems are suitable for being directly mapped onto systolic architectures, as they require local data and local interconnections between the processing elements (PEs).

Definition 1 ***Uniform recurrence equation system***
A system of uniform recurrence equations (SURE) is a collection of $s \in N$ equations of the form (2) and input equation of the form (3):

$$V_i(z) = f_i(V_1(z - \theta_{i_1}), \ldots, V_k(z - \theta_{i_k})) \tag{2}$$
$$V_i(z_i^j) = v_i^j, j \in \{1, ..., l_i\} \tag{3}$$

where

- $V_i : D \to R$. $V_i, i \in \{1, \ldots, s\}$ *are variable names belonging to a finite set* V. *Each variable is indexed with an integral index, whose dimension,* n *(called the index dimension), is constant for a given SURE (in practice this is usually 2 or 3).*
- $z \in D$, *where* $D \subseteq \mathcal{Z}^n$ *is the domain of the SURE.*
- v_i^j *is a scalar constant (input),* $z_i^j \in D_{inp}$, *where* $D_{inp} \subseteq \mathcal{Z}^n$ *is the domain of the inputs.*
- $\theta_{i_1}, \ldots, \theta_{i_k}$ *are vectors of* Z^n *and are called* dependence vectors *of the* SURE.
- $V_i(z)$ *does not appear on the right-hand side of the equation.*
- $f_i : R^s \to R$.

Informally, a SURE (as well as the associated dependence graph) can be seen as a multidimensional systolic array, where the points of the domain D are the PEs of the array and the communication channels are determined by the dependencies. In this context, a transformation applied to the system which preserves the number of domain points and the dependencies leads to a computationally equivalent system. The goal of such a transformation is to obtain a system where one of the indices can be interpreted as the *time index* and the others as *space-indices*.

The form of the SURE, used as a starting point, has a considerable impact on the result of the design.

If we start for example with the following algorithm:

```
$c_j = 0, \forall j, 0 \leq j \leq m + n - 2$
for $i = 0$ to $n - 1$
    for $j = i$ to $i + m - 1$
        $c_j = c_j + a_i * b_{j-i}$,
```

then we can formulate the uniform recurrence equation system (4)-(5). We mention that there are well-known uniformisation techniques (see [13]) to deduce the SURE from the given algorithm, but unfortunately they cannot be applied in a fully automatic way. Let us therefore consider the system (4)-(5) as starting point.

Equations:

$$\begin{cases} C_{i,j} &= C_{i-1,j} + B_{i-1,j-1} * A_{i,j-1} \\ B_{i,j} &= B_{i-1,j-1} \\ A_{i,j} &= A_{i,j-1}, \end{cases} \tag{4}$$

where $0 \le i \le n-1$, $i \le j \le i+m-1$.

Input equations:

$$\begin{cases} B_{-1,i} &= b_{i+1}, & -1 \le i \le m-2 \\ C_{-1,i} &= 0, & 0 \le i \le m-1 \\ C_{i-1,i+m-1} &= 0, & 1 \le i \le n-1 \\ A_{i,i-1} &= a_i, & 0 \le i \le n-1. \end{cases} \tag{5}$$

The results are considered to be the values of the following variables:

$$c_i = \begin{cases} C_{i,i}, & 0 \le i \le n-2 \\ C_{n-1,i}, & n-1 \le i \le n+m-2. \end{cases}$$

Fig. 1 presents the dependence graph associated to the SURE (4)-(5), when $n = 3, m = 4$.

Each of the points of the domain $D = \{(i,j) | 0 \le i \le 2, i \le j \le i+3\}$ corresponds to a computation, while the arrows represent the data dependencies. The placement of the input values can also be read from the figure, although this was determined after the computation of the timing function. The small dots between the points $(i,j), (i+1, j+1)$ of the domain D indicate a delay, that is, the b values need two time steps to move from point (i,j) to $(i+1, j+1)$. Now we have the dependencies shown in Table 1.

The space-time transformation method consists basically of four main steps:

1. the formulation of the problem as a system of uniform recurrence equations,
2. finding one (ore more) adequate *timing function(s)*
   – the timing function determines the time instant when the computation takes place,
3. finding one (ore more) adequate *allocation function(s)* corresponding to a certain timing function

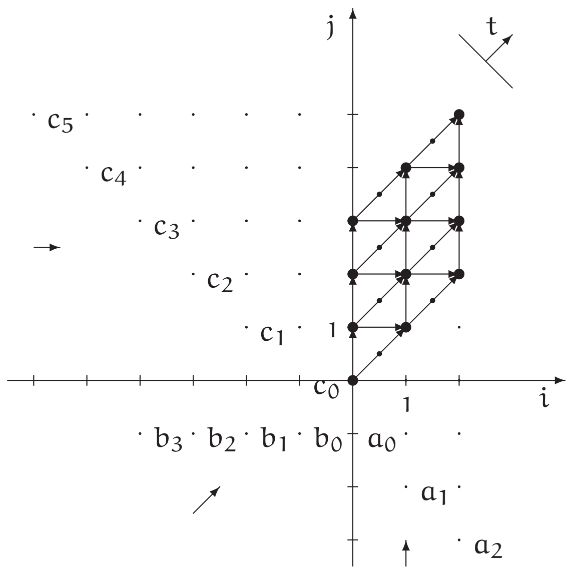

Figure 1: Dependence graph for polynomial multiplication ($n = 3, m = 4$): data dependencies, placement of input data

– the allocation function determines the place (that is the processor) where the computation is performed,

4. application of the space-time transformation.

Linear (and affine) transformations are most commonly used for both, the timing and the allocation function, because thus we obtain a linear space-time transformation, which preserves the dependencies between the computations. Moreover, if the transformation is *unimodular*, then it has the advantage that it preserves the number of points in the domain, and in addition it admits an integral inverse. However, it is not mandatory to use unimodular transformations.

Given a SURE, the next step is to obtain a possible linear (or affine) timing function, which should be positive and should preserve the data dependencies. A natural requirement is that in order to be able to perform the computations of $V_i(z)$, its arguments should have been computed before. If such a function exists, then we say that the SURE is *computable*.

Equation	lhs	rhs	Dependence vector
(14)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1,1)$
(16)	$A1_{i,j}$	$A1_{i,j-1}$	$(0,1)$
(18)	$B1_{i,j}$	$B1_{i,j-1}$	$(0,1)$
(20)	$A2_{i,j}$	$A2_{i-1,j}$	$(1,0)$
(22)	$B2_{i,j}$	$B2_{i-1,j}$	$(1,0)$

Table 1: Dependence vectors

The previously mentioned constraints build a system of inequalities. Any of its solutions gives an adequate timing function.

We might also want to minimise the computation time of the system. In this case, we obtain the timing function:

$$t(i,j) = i + j.$$

The timing function was determined according to the method described in [14, 2] (we avoid to detail the computations here).

In order to get an adequate allocation function for a given timing function, the condition that should hold (we also call it *general constraint*) can be intuitively expressed in the following way: two different computations performed at the same time-step should not be mapped onto the same processor. This means that the linear part P of the allocation function should not be parallel to the direction T corresponding to the timing function (in our case $T = (1,1)$).

For the previously obtained timing function, we get the following allocation functions, which satisfy the above-mentioned condition, moreover, the resulting space-time transformation is unimodular:

$$\begin{aligned} p(i,j) &= j - i \\ p(i,j) &= i \\ p(i,j) &= j. \end{aligned}$$

If we choose the allocation function $p(i,j) = j - i$, after the application of the space-time transformation we obtain the linear systolic array depicted in Fig. 2.

With the allocation function $p(i,j) = i$, we get the systolic array from Fig. 3, while with $p(i,j) = j$ the array from Fig. 4 is obtained. The placement of the inputs is also depicted in the figures. In case of Fig. 3, the structure of the array respectively the transition function is also shown.

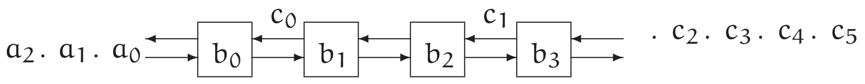

Figure 2: Systolic array for polynomial multiplication (the allocation function $p(i, j) = j - i$ was used)

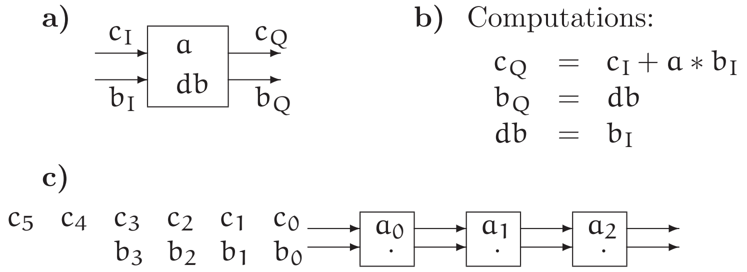

Figure 3: Unidirectional systolic array for polynomial multiplication (the allocation function $p(i, j) = i$ was used for the projection) a) structure of a PE, b) transition function, c) structure of the array and placement of the input values

The data-flow in the arrays of Fig. 3 and Fig. 4 is unidirectional. In the case of the array of Fig. 3 the elements of the result appear after n time steps (where n is the number of PEs) as the output of the PE on the right edge of the array, while in the case of the array from Fig. 4 the results are computed in the local memories of the PEs.

The systolic array depicted in Fig. 2 is bidirectional, but the PEs work alternately and they only perform useful computation at each second time step. There are some well-known techniques to transform such arrays into a more efficient one. Some ideas are presented in [11].

2.2 Functional approach

We have seen that the space-time transformation methodology works with the whole index space. Due to the symmetric structure of the systolic array, this leads to many repetitions in the design process. In case of our functional-based approach to the systolic array design, however, we only have to analyse the

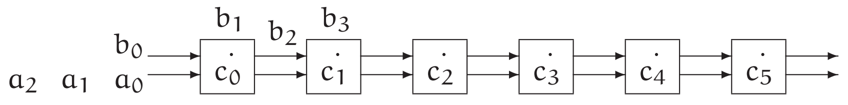

Figure 4: Systolic Array for Polynomial Multiplication (the allocation function $p(i,j) = j$ was used)

behaviour of the first processor, exploiting the idea that the tail-array works in the same way as the original one, solving actually the same kind of problem of a smaller size.

The *functional view* (or inductive view) of systolic arrays is shown in Fig. 5: informally, a linear systolic array with n PEs can be seen as a device, that is composed of a head processor (PE_0), connected to a tail-array, which is an identical array of size $n-1$.

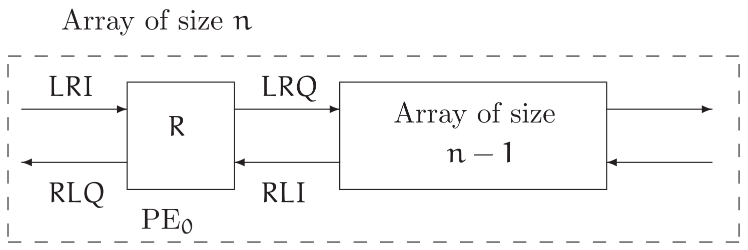

Figure 5: Informal view of a linear systolic array – functional approach

The arrows indicate the direction of the data-flow, from left to right (LR) or from right to left (RL). The letter I stands for *input channels*, Q indicates the *output channels* and R stands for the *internal state registers* (also called local memory).

At each time step the PEs update their internal state (the values of the output channels, respectively that of the internal registers) in function of the input, respectively the value of the internal state registers in the previous time step. The computations performed by a PE are given by the so called *transition function.*

The global input is fed step by step into the array through the input channels of the PEs on the edge, while the result appears at one or more output channels of the marginal PEs (in some cases the result may be computed in the internal state registers as in the case of the systolic array in Fig. 4).

A fundamental step of the design method is the formal analysis of the different systolic array types. In case of each class of systolic arrays, we characterise by a recursive description the class of functions which can be realised by such type of arrays.

Then, by equational rewriting, the expression of the list function which must be realised is transformed into an expression having the required structure. The resulting expression reveals the scalar function, which must be implemented by each individual processor.

The linear systolic arrays can be one- or bidirectional, depending on the direction of the data-flow. A typical subclass of systolic arrays is that, where the input data passes through the array unchanged.

The input or the output data-flow can be delayed or not, the arrays may have more simple building blocks, that is PEs without internal state (also called *combinatorial PEs*), or PEs having constant or variable internal state registers (*local memory*).

Let us consider for example unidirectional systolic arrays with constant internal state registers and delayed input. An example for such an array is depicted in Fig. 3.

We use the following notations (same as in [6, 9]):

- We denote by X_i (where $i \in \mathbb{Z}$) the *infinite list* $\langle x_i, x_{i+1}, x_{i+2} \ldots \rangle$. X stands for X_0. $X_{n,n+m}$ (where $n \in \mathbb{Z}$ and $m \in \mathbb{N}$) denotes the finite list having $m+1$ elements: $\langle x_n, x_{n+1}, \ldots x_{n+m} \rangle$.
- We will denote by a^n the list of n elements all equal to a and by a^∞ the infinite constant list with all elements equal to a.
- For any list $X = \langle x_0, x_1, \ldots, x_n, \ldots \rangle$, we denote by $H[X] = x_0$ the *head* of it, and by $T[X] = \langle x_1, \ldots, x_n, \ldots \rangle$ the *tail* of it.
- The k^{th} *tail* respectively *head* of X:
  $T_k[X] = \langle x_k, x_{k+1}, \ldots, x_n, \ldots \rangle$, for $k \geq 0$ is obtained by iterating T k times. Note that $T_1 = T$. By convention $T_0[X] = X$.
  T_k, for $k < 0$ is obtained by iterating T_{-1} $|k|$ times,
  where $T_{-1}[X_i] = X_{i-1}$
  $H_k[X] = H[T_k[X]]$ gives the $(k+1)^{\text{th}}$ element of X (thus $H_0 = H$).
- The *prefix* of order n of a list is $P_n[X] = \langle x_0, \ldots, x_{n-1} \rangle = X_{0,n-1}$.
- The *concatenation* of two lists is denoted by "$\smile$":
  $\langle a_0, a_1, \ldots, a_k \rangle \smile X = \langle a_0, a_1, \ldots, a_k, x_0, x_1, \ldots \rangle$.
  The first operand must be finite, but the second may also be infinite. We also use "$\dot{\smile}$" for *prepending* a scalar to a (finite or infinite) list: $a \dot{\smile} X = \langle a \rangle \smile X$.

- We use (as in the theory of cellular automata) a special *quiescent symbol* “\$” in order to encode the “blank” values.

The list function $\vec{f}$ in (6) is the *list extension* of the scalar function f (we obtain $\vec{f}[X]$ by applying the function f onto the elements of X). In the same time the expression (6) characterizes the transition function of one PE [10].

$$\vec{f}[x \dot{\smile} X] = f[x] \dot{\smile} \vec{f}[X] \tag{6}$$

Note that the syntactic restriction to one argument (and one value) is not essential. X could also represent a multiple list (composed of a finite k number of lists):

$$\begin{aligned} X &= \langle w^1 \dot{\smile} W^1,\ w^2 \dot{\smile} W^2, \dots w^k \dot{\smile} W^k \rangle^T = \\ &= \langle w^1, w^2, \dots w^k \rangle^T \dot{\smile} \langle W^1, W^2, \dots W^k \rangle^T \ . \end{aligned}$$

Thus a function having multiple list arguments can be seen as a function with one single list-argument (even if we do not mention it explicitly).

The functioning of unidirectional systolic arrays with constant local memory and delayed input is characterised by (7)-(8) (see [10]), where n is the number of PEs (the size-parameter of the problem), X is the input list, the values of $U_{0,n-1}$ correspond to the constant values of the local memory variables and can be considered as parameters of the problem. Y denotes the global output-list which collects the (partial) results, while $Y^0 = (y^0)^\infty$ gives the list of initial values, which contribute to the computation of the results (usually the same y^0 value is introduced repeatedly).

$$\begin{aligned} F_{Q_{0,n-1}}[n, X] &= \vec{f}_{q_{n-1}}[X_{-(n-1)}, X_{-n}, F_{Q_{0,n-2}}[n-1, X]] && (7) \\ F_{q_0}[1, X] &= \vec{f}_{q_0}[X, X_{-1}, Y^0], && (8) \end{aligned}$$

where the list function $\vec{f}_q[\langle X, X', Y \rangle]$ satisfies property (6).
Given $F_{U_{0,n-1}}[n, X]$, our task is to find $Q_{0,n-1}$, which is a permutation of $U_{0,n-1}$, y^0 (such that $Y^0 = (y^0)^\infty$) and the transition function, denoted by f such that (7)-(8) should hold.

Let us consider again the problem of polynomial multiplication. The coefficients of one polynomial will be matched with the finite list of parameters (let us choose for this purpose the n coefficients of polynomial A, that is the list $A_{0,n-1}$), while the coefficients of the other polynomial will form the input list. We get an infinite input list by adding an infinite number of 0 elements to the list of coefficients $(B_{0,m-1} \smile 0^\infty)$. We will also insert a number of $n-1$ elements of 0 in front of the list in order to describe the problem according to the idea depicted in Fig. 6.

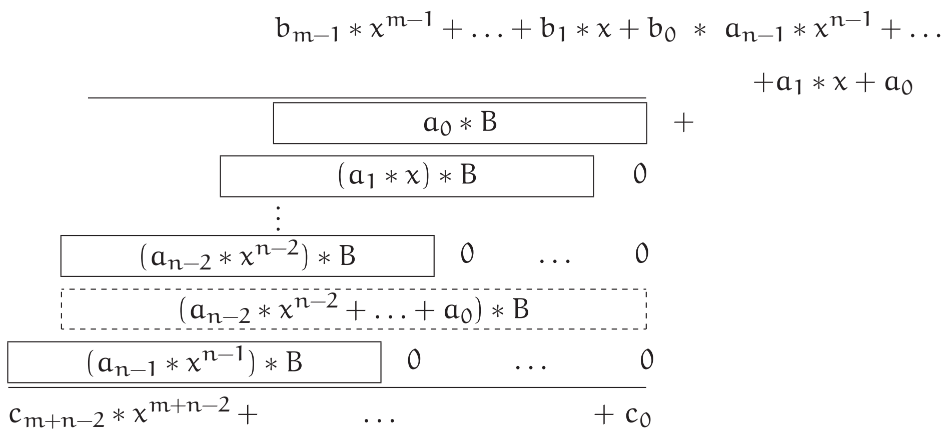
$b_{m-1} * x^{m-1} + \ldots + b_1 * x + b_0 \;*\; a_{n-1} * x^{n-1} + \ldots$
$+a_1 * x + a_0$
$a_0 * B$
$+$
$(a_1 * x) * B$
0
$(a_{n-2} * x^{n-2}) * B$
$0 \quad \ldots \quad 0$
$(a_{n-2} * x^{n-2} + \ldots + a_0) * B$
$(a_{n-1} * x^{n-1}) * B$
$0 \quad \ldots \quad 0$
$c_{m+n-2} * x^{m+n-2} + \quad \ldots \quad + c_0$

Figure 6: Polynomial multiplication

The coefficients of the product will be the first $n+m-1$ elements of the list C:
$C = \langle c_0, c_1, \ldots \rangle$, where

$$c_i = \sum_{j=0}^{n-1} a_j * b_{i-j}, \forall i, i = 0, 1, \ldots n+m-2, c_i = \$, \forall i, i \geq n+m-1$$

Using the more concise list notation this means:

$$C = \sum_{j=0}^{n-1} a_j * B_{-j} \tag{9}$$

(where $a*(b \,\dot{\smile}\, B) = \langle a*b \rangle \,\dot{\smile}\, (a*B)$)

We can write that

$$\begin{aligned} F_{A_{0,n-1}}[n, B] &= \textstyle\sum_{j=0}^{n-1} a_j * B_{-j} = a_{n-1} * B_{-(n-1)} + \sum_{j=0}^{n-2} a_j * B_{-j} = \\ &= a_{n-1} * B_{-(n-1)} + F_{A_{0,n-2}}[n-1, B], \end{aligned}$$

which is of the form (7). We get by simple projection the part of the transition function corresponding to the computation of the c (partial) results: $f[b, db, a, c] = a*b + c$. The rest of the transition function is already known.

By analysing condition (8) we get: $F_{a_0}[1, B] = \vec{f}[a_0, X, X_{-1}, C^0]$, that is $a_0 * B = a_0 * B + C^0 \Rightarrow c^0 = 0$ and $C^0 = 0^\infty$.

We got the solution of Fig. 3 with much less effort than in the case of the space-time transformation method. The systolic array obtained is also appropriate for integer multiplication, only a local memory register of variable value should be added to each PE, in order to preserve the carry [10].

The automatic synthesis of the arrays shown in Fig. 2 and Fig. 4 is similarly simple. However, because these have a different structure, we have to start from another description of the problem, while in the case of the space-time transformation method we obtained the three different solutions starting from the same recurrence equation. However, if we would like to design a systolic array with predefined properties, this is not a drawback at all.

3 Online systolic multiplication

In this section we describe an online systolic array, the functional-based design of which was detailed in a former paper [6]. After outlining the results, we present how such an array could have been synthesized using the space-time transformation methodology.

Online arrays are an important special subclass of bidirectional arrays. They are characterised by the fact that they begin to provide the first result after a constant number of time steps (regardless of the number of PEs). This feature make them very useful for solving real time problems, where the response time is a critical factor.

The array receives the input data through the first PE and the elements of the result leave the array through the same PE.

3.1 Solutions obtained using the functional approach

We have presented the design of such systolic arrays in [6], and we used as case study the design of online arrays for polynomial multiplication, respectively the multiplication of multiple precision integers. That is why we do not detail the design process here; in the sequel, we will only outline its main steps.

Step 0: formal analysis of the systolic array with the given properties
We have analysed the behaviour of specific online systolic arrays with input list X, where the input X' of the tail-array is $T_k[X]$ for some fixed k, thus if the array computes the function $F[X]$, then the tail-array will compute $F[T_k[X]]$.

Such a behaviour of the input can be achieved by including into the internal state a "state variable" s with values from $\{0 = \$, 1, 2, \ldots, k + 2\}$, and the following assignments for s:

$$s \;:=\; \begin{cases} s, & x = \$ \;\vee\; s = k+2 \\ s+1, & x \neq \$ \;\wedge\; s < k+2 \end{cases}$$

The PE will send the x values to the next PE if $s \geq k$, otherwise a \$ value will be passed.

The functioning of such an array (for $k = 2$) is characterised by (10), where G denotes the function which computes the internal state of the array (the internal state includes besides the computation of the output values the values of the local memory variables, too), and f_y is the part of the transition function which computes the (partial) results:

$$T_4[F[X]] = \vec{f_y}[T_4[X],\; T_3[G[X]],\; F[T_2[X]]]. \tag{10}$$

Step 1: formulation of the problem as a functional program
The two polynomials are represented by the list of coefficients completed with an infinite number of redundant zeroes. The input list is the multiple list compound of these two lists.

We assume as known the scalar operations "$\ddot{+}$" and "$\ddot{*}$" in the ring of the coefficients. We will use the functional definition of the simple operations to *unfold* the expression "$A * B$", until we get an equation of the form (10).

Some definitions (we transformed the notations used in mathematics to our list-notation in a very simple, natural way):

- addition of a scalar with a polynomial: $a \dot{+} (b \dot{\smile} B) = (a \ddot{+} b) \dot{\smile} B$
- addition of polynomials: $(a \dot{\smile} A) + (b \dot{\smile} B) = (a \ddot{+} b) \dot{\smile} (A + B)$
- multiplication of a scalar with a polynomial:
  $a \dot{*} (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} (a \dot{*} B)$
- multiplication of polynomials:
  $(a \dot{\smile} A) * (b \dot{\smile} B) = (a \ddot{*} b) \dot{\smile} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} (A * B)))$.

Step 2: unfolding
Unfolding consists in extracting repetitively the elements of the result list, beginning with the first one, by using the functional definitions of the list functions and a few simple unfolding rules, presented in [6].

After the unfolding of the first four elements of the expression $A * B$, we get the following result:

$$\begin{aligned}
& A * B = \\
& = \dots \\
& = \langle \quad a_0 \ddot{*} b_0, \\
& \qquad a_0 \ddot{*} b_1 \ddot{+} b_0 \ddot{*} a_1, \\
& \qquad a_2 \ddot{*} b_0 \ddot{+} a_1 \ddot{*} b_1 \ddot{+} a_0 \ddot{*} b_2), \\
& \qquad a_3 \ddot{*} b_0 \ddot{+} a_2 \ddot{*} b_1 \ddot{+} a_1 \ddot{*} b_2 \ddot{+} a_0 \ddot{*} b_3 \rangle \smile \\
& \smile ((a_0 \dot{*} B_4) + (b_0 \dot{*} A_4) + \\
& \quad +(a_1 \dot{*} B_3) + (b_1 \dot{*} A_3) + (A_2 * B_2))
\end{aligned}$$

From here we can write the equality of the form (10):

$$T_4[A * B] = + \begin{cases} H_0[A] \dot{*} T_4[B] \\ H_0[B] \dot{*} T_4[A] \\ H_1[A] \dot{*} T_3[B] \\ H_1[B] \dot{*} T_3[A] \\ T_2[A] * T_2[B] \end{cases},$$

respectively the first four elements of the output.

Step 3: the elements of the resulted expression are associated to the corresponding elements of the systolic array (using already specified rewrite rules). The head and tail functions H_i and T_i are realised by adding some suitable *static* respectively *transition* variables to the internal state. The list having (almost) all elements equal to H_i is realized by a "static" variable h_i having the assignment: $h_i := \begin{cases} x, & \text{if } s = i \\ h_i, & \text{if } s \neq i \end{cases}$

Let us also consider the "transition" variables z_0, z_1, z_2, z_3 having the assignments: $z_0 = z_1,\ z_1 = z_2,\ z_2 = z_3,\ z_3 = x$. In the expression of $T_4[F[X]]$, the subexpression $T_4[X]$ will be realized by the expression x, and each $T_i[X]$ will be realized by the expression z_i (for $0 \leq i \leq 3$).

We denote the input channels by xa and xb, and the corresponding static and transition variables by ha_i, hb_i, respectively za_i, zb_i, as shown in Fig. 7.

According to the rules mentioned above, the expression on the right–hand side is projected into:

$$\begin{aligned}
& (ha_0 \ddot{*} xb) \ddot{+} (hb_0 \ddot{*} xa) \ddot{+} \\
& \ddot{+} (ha_1 \ddot{*} zb_3) \ddot{+} (hb_1 \ddot{*} za_3) \ddot{+} y'
\end{aligned}$$

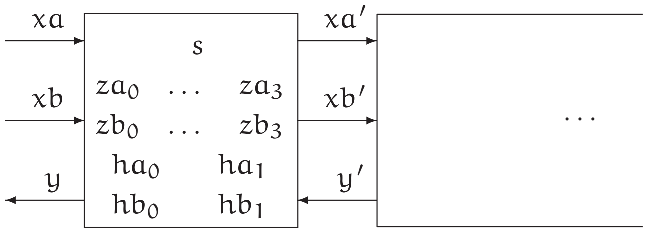

Figure 7: Online systolic array for polynomial multiplication

The expressions representing the computation of the first 4 elements are found in the same way.

Thus, the part of the transition function describing the assignment for the output channel y (that is, the computation of the result) is:

$$\begin{cases} \$ & s = \$ = xa \\ xa \ddot{*} xb & s = \$ \neq xa \\ hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & s = 1 \\ ha_1 \ddot{*} hb_1 \ddot{+} & \\ \quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & s = 2 \\ hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\ \quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb & s = 3 \\ hb_1 \ddot{*} za_3 \ddot{+} ha_1 \ddot{*} zb_3 \ddot{+} & \\ \quad \ddot{+} hb_0 \ddot{*} xa \ddot{+} ha_0 \ddot{*} xb \ddot{+} y' & s = 4 \end{cases}$$

The rest of the transition function, containing the computation of variables $s, ha_i, hb_i, za_i, zb_i, xa', xb'$ is known.

We can use the same kind of array for the multiplication of arbitrary large integers again, by adding a register of variable value to each PE [6].

3.2 Solution using the space-time transformation method

None of the systolic arrays obtained from the SURE (4)–(5) is an online one. As we already mentioned, the result is significantly influenced by the form of the SURE used as starting point. Consequently, we need another formulation

of the problem, which again requires some intuition. We use the following notation:
In the sequel let the values of A_i be equal to a_i if $0 \le i \le n-1$, otherwise 0. In the same way $B_j = b_j$, if $0 \le j \le m-1$, otherwise 0.

$$A*B = \underbrace{A_0*B_0}_{C_{0,0}} + (\underbrace{A_0*B_1 + A_1*B_0}_{C_{0,1}})*x + (\underbrace{A_0*B_2 + \overbrace{A_1*B_1}^{C1,1} + A_2*B_0}_{C_{0,2}})*x^2 +$$

$$+\,(\underbrace{A_0*B_3 + \overbrace{A_1*B_2 + A_2*B_1}^{C_{1,2}} + A_3*B_0}_{C_{0,3}})*x^3 + \dots$$

Generally:
$\forall i,j:\ 0 \le i \le j;\ i+j \le m+n-2$

$$C_{i,j} = \begin{cases} A_i*B_i & i=j \\ A_i*B_j + A_j*B_i & j=i+1 \\ A_i*B_j + A_j*B_i + C_{i+1,j-1} & j>i+1 \end{cases} \tag{11}$$

The result: $c_k = C_{0,k}$, $\forall k,\ 0 \le k \le m+n-2$.

3.2.1 Uniformisation of the recurrence equation

In equation (11), A_i is needed in the computation of $C_{i,j}$ for all values of j, $i \le j \le m+n-2-i$, this means a broadcast of A_i. Similarly, A_j is needed in the computation of $C_{i,j}$, $\forall i, 0 \le i \le m+n-2-j$. A common method to eliminate broadcast is to pipeline the given value through the nodes where it is needed (see [13]). Thus, we replace A_i with a new variable $A1_{i,i}$, and pipeline it in the direction $(i,j) \to (i,j+1)$. A_j will be replaced by the variable $A2_{0,j}$ and pipelined through the direction $(i,j) \to (i+1,j)$. B_i and B_j will be replaced in the same way with $B1$ and $B2$, respectively.

We obtain the following uniform recurrence equation:

$$\forall i,j:\ 0 \le i \le j;\ i+j \le m+n-2$$

$$C_{i,j} = \begin{cases} A2_{i,j}*B2_{i,j} & j=i & (12) \\ A1_{i,j}*B2_{i,j} + A2_{i,j}*B1_{i,j} & j=i+1 & (13) \\ A1_{i,j}*B2_{i,j} + A2_{i,j}*B1_{i,j} + C_{i+1,j-1} & j>i+1 & (14) \end{cases}$$

$$A1_{i,j} = \begin{cases} A_i & j=i & (15) \\ A1_{i,j-1} & j>i & (16) \end{cases}$$

$$B1_{i,j} = \begin{cases} B_i & j = i \quad (17) \\ B1_{i,j-1} & j > i \quad (18) \end{cases}$$

$$A2_{i,j} = \begin{cases} A_j & i = 0 \quad (19) \\ A2_{i-1,j} & i > 0 \quad (20) \end{cases}$$

$$B2_{i,j} = \begin{cases} B_j & i = 0 \quad (21) \\ B2_{i-1,j} & i > 0 \quad (22) \end{cases}$$

Note that equations (15), (17), (19), (21) are input equations of the form (3).

Now the input A_i appears in input equation (15) and (19), too. B_i also appears in two input equation. This would mean that we have to input the coefficients of the polynomials A and B twice.

This can be avoided by changing input equation (15) with

$$A1_{i,j} = A2_{i,j} \qquad j = i. \tag{23}$$

In the same way, we change (17) by:

$$B1_{i,j} = B2_{i,j} \qquad j = i. \tag{24}$$

Table 2. shows the dependencies of the SURE.

Equation	lhs	rhs	Dependence vector
(14)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1, 1)$
(16)	$A1_{i,j}$	$A1_{i,j-1}$	$(0, 1)$
(18)	$B1_{i,j}$	$B1_{i,j-1}$	$(0, 1)$
(20)	$A2_{i,j}$	$A2_{i-1,j}$	$(1, 0)$
(22)	$B2_{i,j}$	$B2_{i-1,j}$	$(1, 0)$
(14)	$C_{i,j}$	$A1_{i,j}, A2_{i,j}, B1_{i,j}, B2_{i,j}$	$(0, 0)$
(23)	$A1_{i,j}$	$A2_{i,j}$	$(0, 0)$
(24)	$B1_{i,j}$	$B2_{i,j}$	$(0, 0)$

Table 2: Dependence vectors

Note that the dependencies for $A1$ and $B1$ respectively $A2$ and $B2$ are the same. In the following we will only reason about $A1$ and $A2$; $B1$, respectively $B2$, can be handled similarly.

3.2.2 Finding an adequate timing function

According to the method presented in [2], we are looking for affine timing functions with the same linear part for each variable V of the SURE (12)–(22) of the form $t_V = x * i + y * j + z_V$.

For each dependence of Table 2 of the form $V_i(z) \leftarrow V_j(z')$, we are writing the dependency constraint of the form $t_{V_i}(z) > t_{V_j}(z')$. We get:

$$\begin{array}{lcllc} C_{i,j} & \leftarrow & A1_{i,j} & \Rightarrow & t_C(i,j) > t_{A1}(i,j) \\ C_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow & t_C(i,j) > t_{A2}(i,j) \\ C_{i,j} & \leftarrow & C_{i+1,j-1} & \Rightarrow & t_C(i,j) > t_C(i+1,j-1) \\ A1_{i,j} & \leftarrow & A1_{i,j-1} & \Rightarrow & t_{A1}(i,j) > t_{A1}(i,j-1) \\ A2_{i,j} & \leftarrow & A2_{i-1,j} & \Rightarrow & t_{A2}(i,j) > t_{A2}(i-1,j) \\ A1_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow & t_{A1}(i,j) > t_{A2}(i,j) \end{array} \tag{25}$$

From the conditions marked with (25) and the computation time minimization condition we get the following system of inequalities:

$$\begin{cases} z_c > z_{A1} \\ z_c > z_{A2} \\ y - x > 0 \\ y > 0 \\ x > 0 \\ z_{A1} > z_{A2} \\ x + y + z_C + z_{A1} + z_{A2} \longrightarrow \text{minimal} \end{cases} \tag{26}$$

We also need the constraint that the time function is positive on the domain. Then from (26) we get the solution:

$$\begin{cases} x = 1 \\ y = 2 \\ z_{A2} = 0 \\ z_{A1} = 1 \\ z_C = 2 \end{cases}$$

The time functions are the following:

$$\begin{array}{lcl} t_C(i,j) & = & i + 2j + 2 \\ t_{A1}(i,j) & = & t_{B1}(i,j) = i + 2j + 1 \\ t_{A2}(i,j) & = & t_{B2}(i,j) = i + 2j \end{array} \tag{27}$$

The common linear part of the time functions is $T = (1, 2)$.

3.2.3 Possible allocation functions

Given the timing functions found in section 3.2.2, we are looking for affine allocation functions with the same linear part for each variable V of the SURE (12)–(22) of the form $p_V = \alpha * i + \beta * j + \gamma_V$. The common linear part of the allocation functions is $P = (\alpha, \beta)$. The *general constraint* in our case is:

$$\frac{\alpha}{\beta} \neq \frac{1}{2}. \tag{28}$$

In Table 2 one can look for the dependence vector corresponding to a certain variable. That is $(-1,1), (0,1)$ and $(1,0)$ for variables C, $A1$ and $A2$, respectively.

For a variable V and a corresponding dependence vector θ_V, the dataflow-direction is $(T * \theta_V, P * \theta_V) = ((1,2) * \theta_V, (\alpha, \beta) * \theta_V)$, where the component $T * \theta_V$ indicates the "speed" of variable V, while the component $P * \theta_V$ shows the direction of the V values.

According to the *weak conditions* (only for dependencies of the form $V(z) \leftarrow V(z'))$, the node z' should be "close enough" to z, such that $V(z')$ can arrive to the required place in $t_V(z) - t_V(z')$ steps. The conditions are:

$$\begin{array}{lll} C_{i,j} \leftarrow C_{i+1,j-1} & \Rightarrow |p_C(i,j) - p_C(i+1,j-1)| & \leq t_C(i,j) - t_C(i+1,j-1) \\ A1_{i,j} \leftarrow A1_{i,j-1} & \Rightarrow |p_{A1}(i,j) - p_{A1}(i,j-1)| & \leq t_{A1}(i,j) - t_{A1}(i,j-1) \\ A2_{i,j} \leftarrow A2_{i-1,j} & \Rightarrow |p_{A2}(i,j) - p_{A2}(i-1,j)| & \leq t_{A2}(i,j) - t_{A2}(i-1,j) \end{array} \tag{29}$$

In the case of dependencies of the form $V_i(z) \leftarrow V_j(z'), i \neq j$, we can write the so-called *strong dependencies* of the form (30).

$$\|p_{V_i}(z) - p_{V_j}(z')\| = \left\lfloor \frac{t_{V_i}(z) - t_{V_j}(z')}{T * \theta_{V_j}} \right\rfloor P * \theta_{V_j} \tag{30}$$

In our case, these are:

$$\begin{array}{lllll} C_{i,j} \leftarrow A1_{i,j} & \Rightarrow & p_C(i,j) - p_{A1}(i,j) & = & \left\lfloor \frac{1}{2}(t_C(i,j) - t_{A1}(i,j)) \right\rfloor \beta \\ C_{i,j} \leftarrow A2_{i,j} & \Rightarrow & p_C(i,j) - p_{A2}(i,j) & = & (t_C(i,j) - t_{A2}(i,j))\alpha \\ A1_{i,j} \leftarrow A2_{i,j} & \Rightarrow & p_{A1}(i,j) - p_{A2}(i,j) & = & (t_{A1}(i,j) - t_{A2}(i,j))\alpha \end{array} \tag{31}$$

From (29) we get:

$$\begin{cases} |\beta - \alpha| & \leq & 1 \\ |\beta| & \leq & 2 \\ |\alpha| & \leq & 1 \end{cases} \tag{32}$$

From (31) we get:

$$\begin{cases} \gamma_C - \gamma_{A1} & = & 0 \\ \gamma_C - \gamma_{A2} & = & \alpha \\ \gamma_{A1} - \gamma_{A2} & = & \alpha \end{cases} \tag{33}$$

From conditions (28) and (32) we get the set of solutions for α and β:

$$(\alpha, \beta) \in \{(-1,-1), (-1,0), (0,-1), (0,0), (0,1), (1,0), (1,1)\} \tag{34}$$

In (34) the first and the last three solutions are symmetric and the solution $(\alpha, \beta) = (0,0)$ can be excluded because the transformation matrix $\begin{pmatrix} T \\ P \end{pmatrix}$ would be then singular (that means that it would transform some points of D lying on a line into a single point, which is not admitted). Thus we have only three different results:

$$P \in \{(0,1), (1,0), (1,1)\} \tag{35}$$

From (35) and (33) we get three different solutions for adequate allocation functions corresponding to the given timing functions:

$$p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = p_{A2}(i,j) = p_{B2}(i,j) = j \tag{36}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i \\ p_{A2}(i,j) = p_{B2}(i,j) = i-1 \end{cases} \tag{37}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i+j \\ p_{A2}(i,j) = p_{B2}(i,j) = i+j-1 \end{cases} \tag{38}$$

3.2.4 Mappings to different systolic arrays

We apply the space-time transformation onto the SURE (12)–(22) according to the timing functions from (27) and allocation functions from (37). That is:

$$\begin{array}{lllll} t_C(i,j) & = & i+2j+2 & & \\ t_{A1}(i,j) & = & t_{B1}(i,j) & = & i+2j+1 \\ t_{A2}(i,j) & = & t_{B2}(i,j) & = & i+2j \end{array} \qquad \begin{array}{lllll} p_C(i,j) & = & i & & \\ p_{A1}(i,j) & = & p_{B1}(i,j) & = & i \\ p_{A2}(i,j) & = & p_{B2}(i,j) & = & i-1 \end{array}$$

We have chosen this transformation, because this is the one the application of which results in an online array. The transformed SURE:

$$\forall t, p: \; p \geq 0; 3p+2 \leq t \leq -p + 2(m+n) - 2; \frac{t-p}{2} \in \mathcal{Z}$$

$$\overline{C}_{t,p} = \begin{cases} \overline{A2}_{t-2,p-1} * \overline{B2}_{t-2,p-1} & t = 3p+2 \quad (39) \\ \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} & t = 3p+4 \quad (40) \\ \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} + & \\ \quad + \overline{C}_{t-1,p+1} & t > 3p+4 \quad (41) \end{cases}$$

$$\overline{A1}_{t,p} = \begin{cases} \overline{A2}_{t-1,p-1} & t = 3p+2 \quad (42) \\ \overline{A1}_{t-2,p} & t > 3p+2 \quad (43) \end{cases}$$

$$\overline{B1}_{t,p} = \begin{cases} \overline{B2}_{t-1,p-1} & t = 3p+2 \quad (44) \\ \overline{B1}_{t-2,p} & t > 3p+2 \quad (45) \end{cases}$$

$$\overline{A2}_{t,p} = \begin{cases} A_{\frac{t}{2}} & p = 0 \quad (46) \\ \overline{A2}_{t-1,p-1} & p > 0 \quad (47) \end{cases}$$

$$\overline{B2}_{t,p} = \begin{cases} B_{\frac{t}{2}} & p = 0 \quad (48) \\ \overline{B2}_{t-1,p-1} & p > 0 \quad (49) \end{cases}$$

Note that this transformation is not unimodular, for this reason the domain of the system (39)-(49) is sparse (see the $(t-p)/2 \in \mathcal{Z}$ condition). The resulted array can be optimised: by merging two neighbouring PEs, we get the online array presented in Section 3.1.

As a conclusion, in this case it is obvious that we have succeeded to design the same systolic array in a more "elegant" and efficient way using the functional approach.

4 Conclusions

In this paper we have compared two automatic systolic array design methods: the space-time transformation methodology and the functional-based method.

We presented different solutions of a representative problem, using both methods, in order to demonstrate the main characteristics, differences, advantages and eventual disadvantages of the two design methods.

The space-time transformation method is obviously the most widespread methodology, and also the most complex one. However, besides its numerous advantages it also has some drawbacks, too: the formulation of the problem as a SURE may be sometimes of serious difficulty, complex computations on the whole index space (repetitions), in order to find an adequate timing function, a complex linear programming problem has to be solved.

Other methods that imply more simple computations (for example concerning the computation of the timing function [7]) only work for a fixed size.

The most relevant advantage of our functional-based method is, that exploiting the symmetric structure of the systolic array, in fact we only have to analyse the behaviour of the first PE. The method also works for parametrized problems (the size of the problem does not have to be fixed in advance). Moreover, the design process consists basically in the application of rewrite rules, thus its implementation is relatively simple.

For the moment, the method is applicable only to linear systolic arrays; this is, however, for practical reasons (efficiency, reliability and ease of implementation) the most popular class of systolic arrays. It relies on a formal analysis performed in advance, thus it is less general than the space-time transformation method.

The considerations above make us believe that it is worth working on the improvement of the functional-based design method by analysing other classes of systolic arrays, too.

References

[1] J.-M. Delosme, I. C. F. Ipsen. Systolic array synthesis: computability and time cones, in *Parallel algorithms & architectures (Luminy, 1986)*, pp. 295–312, North-Holland, 1986. ⇒ 236

[2] P. Feautrier, Some efficient solutions to the affine scheduling problem, Part I : One-dimensional Time, *Int. J. of Parallel Programming*, **21**, 5 (1992) 313–348. ⇒ 241, 253

[3] J. A. B. Fortes, D. I. Moldovan, Data broadcasting in linearly scheduled array processors, *11th Annual Symp. on Computer Architecture*, 1984, pp. 224–231. ⇒ 236

[4] P. Gribomont, V. Van Dongen, Generic systolic arrays: a methodology for systolic design, *Lecture Notes in Computer Science*, **668**, 1992, pp. 746–761.

[5] T. Jebelean, *Systolic multiprecision arithmetic*, PhD Thesis, RISC-Linz Report 94-37, April 1994.

[6] T. Jebelean, L. Ruff. Functional-Based Synthesis of Systolic Online Multipliers, *Proceedings of SYNASC-05 (International Symposium on Symbolic and Numeric Scientific Computing)*, (eds. D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Ciortaş, A. Abraham, M. Paprzycki),

IEEE Computer Society, 2005, pp. 267–275. ⇒ 235, 236, 237, 244, 247, 248, 250

[7] L. Kazerouni, B. Rajan, R. K. Shyamasundar, Mapping linear recurrence equations onto systolic architectures, *International Journal of High Speed Computing (IJHSC)*, **8**, 3 (1996) 229–270. ⇒ 257

[8] H. W. Nelis, E. F. Deprettere, Automatic design and partitioning of systolic/wavefront arrays for VLSI, *Circuits, systems, and signal processing*, **7**, 2 (1988) 235–251. ⇒ 236

[9] L. Ruff, T. Jebelean, Functional–based synthesis of a systolic array for gcd computation, *Lecture Notes in Computer Science*, **4449**, 2007, pp. 37–54. ⇒ 235, 237, 244

[10] L. Ruff, Functional–based synthesis of unidirectional linear systolic arrays, *Pure Math. Appl.*, **17**, 3–4 (2006) 419–443. ⇒ 235, 237, 245, 247

[11] L. Ruff, Optimisation of bidirectional systolic arrays with sparse input by "folding", *10th Symposium on Programming Languages and Software Tools, SPLST07*, Dobogókő, Hungary, Eötvös University Press (eds. Zoltán Horváth, László Kozma, Viktória Zsók), 2007, pp. 420–432. ⇒ 242

[12] P. Quinton, *The systematic design of systolic arrays* in *Automata Networks in Computer Science*, (eds. F. F. Soulie, Y. Robert, M. Tchuente), Manchester University Press, 1987, pp. 229–260. ⇒ 236

[13] P. Quinton, V. Van Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays, *Journal of VLSI Signal Processing* **1**, 2 (1989) 95–113. 236, 239, 251

[14] P. Quinton, Y. Robert. *Systolic algorithms and architectures*, Prentice-Hall, 1990. ⇒ 235, 236, 237, 241

[15] S. W. Song. *Systolic algorithms: concepts, synthesis, and evolution*, Temuco, CIMPA School of Parallel Computing, Chile, 1994. ⇒ 235, 236, 237

[16] L. Ruff. *Automatic design of systolic arrays: a short survey*. Technical report no. 02-27 in RISC Report Series, University of Linz, Austria, December 2002. ⇒ 236

[17] S. Wolfram. *The Mathematica Book*, 5th edition, Wolfram Media, 2003.

Received: April 28, 2009