A parallelization methodology for reconfigurable systems applied to edge detection.

Juan M. Campos*, Rene Cumplido†, Claudia Feregrino-Uribe‡, Roberto Pérez-Andrade‡
* † ‡Instituo Nacional de Astrofisica Optica y Electronica.
Email: *jcampos, †rcumplido, ‡cferegrino, @inaoep.mx
‡Center for Scientific Research and Higher Education of Tamaulipas.
Email: jrperez@tamps.cinvestav.mx

***Abstract*—In this paper, a novel parallelization methodology is applied to Edge Detection Algorithm (EDA). The proposed methodology is based on a multiprojection approach and on a fusion of processor elements. It eliminates the relationship between problem size and processor array size when using methodologies based on projections. EDA is an interesting problem because its data dependencies and its potential parallelism, besides EDA is used in multiple applications. In this study, multiple versions of the EDA architecture are generated in order to fulfill requirements of throughput and implementation area.**

***Keywords*—Design methodologies; Image processing; Parallel processing; Reconfigurable systems; Parallel architectures.**

I. Introduction

Due to its potential to greatly accelerate a wide variety of applications, reconfigurable computing has become a subject of a great deal in research [1]. Its key feature is the ability to perform computations in hardware while retaining flexibility. Flexibility allows modification of some characteristics of the architecture in a number of different ways resulting on improvements of implementation area, throughput or use of memory resources. Such characteristics are directly dependent on the number of processor elements, the communication network, and the data flows within the architecture. Most of the algorithms presents some parallelism, i.e., it is possible to process them by performing more than one single operation per time unit. In order to execute algorithmic tasks in parallel, hardware with multiple processor elements is necessary. Processor arrays are parallel architectures composed of interconnected processor elements. They present one or more of the following characteristics [26]:

- The architecture is implemented by only a few different components.
- Control and data path are regular structures, then the components are connected in a local and regular way. Long distances or irregular connections are avoided or at least minimized.
- Maximum parallelism is achieved when all components are active at the same time. Input-output requirements are minimized by internal transfers between elements producing several computations for each I/O access.

By using a design methodology, it is possible to achieve processor arrays fulfilling the characteristics proposed in [26]. A design methodology is a tool to generate a mathematical model from an algorithm and its data dependencies. Working on the model is possible to generate transformations to improve throughput and data locality while reducing the implementation area. Such transformations are intended to create a parallel version of the original algorithm. Information provided by a design methodology could be used to create a processor array that implements the desired functionality [14].

A. Design methodologies

Design methodologies are particularly useful when the algorithm includes loops as those used in image processing. In the Polytope model, each block of instructions or loop element is represented as a node inside a polytope and data dependencies as arrows between nodes [16], [2], [19], [10]. Two functions relate nodes, mapping them with execution times and physical resources. Both functions represent *when* and *where* nodes will be executed. These functions are called scheduler and allocator and are determined to take full advantage of the available algorithmic parallelism. Optimizing these functions to produce faster and smaller architectures is the main objective of the Polytope model; however, data dependencies conditions this aim. The function to perform the mapping between nodes and execution times is called *scheduler* [5], [6], [18], [21]. If there exists more than one possible schedule function, the fastest is preferred. Allocation refers to the relationship between nodes in the polytope and processor elements [7], [4], [9]. First option is to assign one processor element to each node inside the polytope; however, the cost in the implementation area could be prohibitive.
The problem of finding the optimal scheduler and allocator has been widely studied. In this work, the proposal is to introduce data conditions into the original problem statement in order to generate a processor array able to be compressed. The proposed methodology is applied to the Edge Detection Algorithm (EDA) [23] [24] [22] in order to generate multiple versions of the hardware architecture. The final set of architectures is suitable for a reconfigurable system, covering options from maximum parallelism and high throughput to minimal implementation area with limited throughput.

B. Edge Detection Algorithm (EDA)

Regarding EDA, multiple and large data dependencies are the main challenge for designing and implementing an efficient architecture. This problem has previously boarded in [22] and [24]. Edge detection is one of the most used

978-1-4673-6180-4/13/$31.00 ©2013 IEEE

operations in image analysis. An edge is defined by a discontinuity in gray level values [23]. Edges are one of the most important visual clues for interpreting images. Edge detection is by far, the most common approach for detecting meaningful discontinuities in the gray level. [17]. The edge detection process outputs an image where edge details appear as the outlines of image objects. Edge detection is commonly used as the first stage in complex processes as feature detection and object recognition. There are many different methods for edge detection such as Sobel filtering, Prewitt filtering, Laplacian of Gaussian filtering, moment-based operators, the Shen and Castan operator and the Canny and Deriche operator [17]. No matter the selected approach, all methods require a high amount of computational power. In this context, taking advantage of the inherent parallelism in algorithms while reducing the implementation area is a key point in design of architectures.

The rest of the paper is organized as follows: In Section II, basics on the Polytope model are presented. In Section III, the extended parallelization methodology is introduced. In Section IV, a representative Edge Detection Algorithm as proposed in [24] is modeled and transformed according to the proposed methodology. Finally, discussion and conclusions close the paper in Sections V and VI.

II. Background

The Polytope model [16][8] is a mathematical tool that allows generation of efficient processor arrays from an algorithm. In Polytope context, each loop iteration in the algorithm is represented by one node within a polytope.

Definition 1 (Polytope): A polytope is an intersection of a finite number of half-spaces. Each of the half-spaces provides a face to the polytope.

The *iteration vector* is the vector formed with values of the indices of all loops surrounding one statement. An iteration vector represents a dynamic instance of a statement appearing in a loop nest. A program comprises a sequence of statements, each statement surrounded by loops in a given order. The set of all the iteration vectors is called the *iteration space* (IS).

Definition 2 (Iteration Space): An iteration space IS is a set. Its elements are valid values for an index vector iv. The iteration space is a discrete, not necessarily finite set.

In this work, (IS) is assumed to be an n-dimensional subset of integers. In the Polytope model context, iteration spaces are formulated as polytopes or even more general as so-called linearly bounded lattices.

Definition 3 (Linearly bounded lattice): A linearly bounded lattice (LBL) denotes an iteration space of the form $IS = \{I \in \mathbb{Z}^n \mid I = Mx + c \wedge Ax \geq b\}$ Where $x \in \mathbb{Z}^l$, $M \in \mathbb{Z}^{n \times l}$, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m \times l}$ and $b \in \mathbb{Z}^m$ denotes the set of integral points within a convex polytope or in case of boundedness within a polytope in $\mathbb{Z}^l$. This set is mapped onto iteration vectors iv using an affine transformation ($iv = Mx + c$).

In this work, it is assumed that the matrix M is square and of full rank.

The set of data dependencies is represented by a dependence matrix D as in Equation 1.

$$D = \left[\ \vec{d_1}, \vec{d_2}, ..., \vec{d_i} \ \right] \tag{1}$$

Where, d_i is the i-dependence in the dependence set.

Using data dependencies and IS characteristics, it is possible to find an optimum execution time for each node I in the IS i.e., an optimum execution order by using an optimization process, for instance, a linear program approach [5], [11], [2] and [3].
The result of the optimization process is a schedule vector $\Lambda \in \mathbb{Z}^{1 \times n}$. The schedule vector is used to generate integer execution hyperplanes, orthogonal to it.

Definition 4 (Hyperplane): A hyperplane is an $(n-1)$-dimensional subspace of an n-dimensional vector space.

All nodes in the same hyperplane can be executed in parallel without affecting the functional behavior of the original algorithm. If it is possible to express a function $\phi(I)$ as a schedule vector as shown in Equation 2 then, the scheduler is said to be linear.

$$\phi(I) = \lfloor \Lambda I \rfloor \qquad \forall I \in IS \tag{2}$$

In Equation 2, the linear schedule vector $\Lambda \in \mathbb{Z}^{1 \times n}$ is such that $\Lambda d_i \geqslant 1$ for all $d_i \in D$. This condition [15] ensures that all data dependencies are preserved under the schedule vector Λ and could be expressed as Equation 3.

$$\phi(I_2^{di}) - \phi(I_1^{di}) \geqslant 1 \tag{3}$$

Where I_2^{di} and I_1^{di} are the destiny and source nodes in *IS* of the data dependence (d_i). In other words, if there exists a data dependence d_i between nodes I_2 and I_1, the condition in 3 ensures that the node I_1 will be executed after the node I_2 when using the schedule vector Λ or the schedule function ϕ. Since the schedule vector Λ defines an execution time for each node in IS, it is possible to calculate the total execution time T_Λ of the IS with Equation 4.

$$T_\Lambda = 1 + max(\phi(I_y) - \phi(I_x)) \tag{4}$$

Where I_y and I_x are any two nodes in the IS. The optimal linear scheduler T_l in Equation 5 is the one that minimizes T_Λ over all possible schedule vectors Λ with the only condition of preserving data dependencies (Equation 3).

$$T_l = min(T_\Lambda) \mid \Lambda \in \mathbb{Z}^{1 \times n}, \Lambda D \geq 1 \tag{5}$$

Definition 5 (Iteration Interval): The *iteration interval* λ is the number of integer execution hyperplanes between the execution of two nodes I.

Integer execution hyperplanes refers to the hyperplanes orthogonal to the schedule vector intersecting at least one node I in the IS. However, there exists a more relevant approach of λ value indicated by w which denotes the number of time steps between the production and the consumption of a datum in nodes I_1 and I_2, respectively. Such datum is intended to fulfill the data dependence i as shown in Equation 6. Value w is equal to $(\lambda - 1)$.

$$\phi(I_2^{di}) - \phi(I_1^{di}) = w \tag{6}$$

From processor elements perspective, w indicates how many memory localities are required for storing the data to fulfill

a data dependence. For instance, a value of $w = 3$ indicates a datum will be required three time steps forward. Because a new datum could be generated at each time step, three memory localities are required.

Additionally to the schedule vector, it is necessary to know in which processor element will be computed each node I. In this work, the selected method for such allocation [11], [3] is the projection [13], [25]. The projection is represented as the vector P. Similar to the schedule vector, the allocation vector P could be expressed as the allocation function $\rho(I)$.

$$\rho(I) = \lfloor \mathrm{PI} \rfloor \qquad \forall I \in IS \tag{7}$$

Function $\rho(I)$ sets a relationship between nodes in the IS and processor elements. For instance, Equation 8 indicates that both nodes I_y and I_x will be executed in the same processor element using the allocation function $\rho(I)$.

$$\rho(I_y) - \rho(I_x) = 0 \tag{8}$$

The projection vector should be carefully selected since a bad choice could be unfavorable in terms of throughput or implementation area. Traditionally, the allocation vector has been proposed by hand requiring multiple tests to select the best option in terms of performance and implementation area [11], [20]. Additionally, under the projection approach, the number of processor elements in the processor array depends on the size of the problem.

Main contributions of this work are the next:

- The allocation vector is automatically obtained and finally the processor array is compressed.
- Different versions of the same architecture are generated providing a wide range of implementation options suitable for a reconfigurable system.
- Memory requirements are considerably smaller when compared with processor arrays generated with traditional approaches.

III. Extended parallelization methodology

In this work, an extended parallelization methodology based on the seminal work of Alain Darte [2], [3] is applied to the EDA problem, previously boarded in [22], [24]. This process provides a number of different architectures for implementing the EDA algorithm while reducing the number of processor elements in the processor array.

The proposed methodology consists of the following stages:

- Algorithmic representation
- Allocation
- Scheduling
- Reducing the number of processor elements

In the proposed methodology, the allocator is firstly obtained. After, by modifying the dependence set, the scheduler is calculated. This strategy is intended to generate the required conditions to compress the processor array.

A. Algorithmic representation

In general, algorithms are firstly presented using mathematical language and after expressed using loop structures for software implementation/simulation. Using a loop representation is useless when the objective is to extract the parallelism because such representations impose a serial execution order. According to [11], it is possible to model any algorithm directly from its mathematical representation to produce a *system of uniform recurrence equations* (SURE) representation avoiding the *loop* translation.

The concept of SURE was introduced in 1967 [12] for modeling regular iterative processes. The idea is to model k functions $a_1(I), a_2(I), ..., a_k(I)$ where each function $a_i(I), i \in [1..k]$ is assumed to be evaluated in all the points I en IS where IS is an integral subset, IS $\subseteq \mathbb{Z}^n$

Using a SURE representation, the concept of data dependency could be represented in a natural manner. A single recurrence equation is of the form:

$$a_1(I) = F_1(a_1(I - d_1), a_1(I - d_2), ..., a_1(I - d_k)) \tag{9}$$

Where $I \in IS$, $d_j | j \in [1..k]$ is an n-dimensional integer vector called iteration vector and F_1 is a single-valued function. If any difference $I - d_j$ is element of $\mathbb{Z}^n$ and each vector d_j is constant, the equation is said to have a uniform dependency. A system of uniform recurrence equations is generally given by

$$a_i(I) = F_i(a_{i_1}(I - d_{i_1}), ..., a_{i_k}(I - d_{i_k})) \quad \forall I \in IS \tag{10}$$

Where $IS \subseteq \mathbb{Z}^n, d_{i_j} \in \mathbb{Z}^n, j \in [1..k]$ any difference $I - d_j \in \mathbb{Z}^n$, and F_i is an arbitrary function. In case of representing algorithms, instead of functions, indexed variables are considered. Such indexed variables are supposed to be defined for each element on $IS = \left\{ (i,j)^T \in \mathbb{Z}^n | 1 \leq i \leq \max_i \wedge 1 \leq j \leq \max_j \right\}$

B. Allocation

In this work, the allocating function is firstly obtained. Allocating function is obtained by a linear programming approach including data dependencies as conditions. This approach produce a processor array that responds to the data flow requirements. The process is similar to the one described in Section II for obtaining the schedule vector.

To achieve it, Equation 11 is proposed where N_P indicates the number of processor elements required to implement the IS using the allocator P.

$$N_P = 1 + max(\rho(I_y) - \rho(I_x)) \tag{11}$$

Where I_y and I_x are any two nodes in the IS. In this proposal, the optimal allocator is the one that minimizes N_P over all possible allocation vectors P such that Equation 3 is fulfilled. This is expressed as Equation 12.

$$N_l = min(N_P) \mid P \in \mathbb{Z}^{1 \times n}, \qquad PD \geqslant 1 \tag{12}$$

In this work, an allocation approach on Equation 3 allows optimization of the use of processor elements. If there exists

a data dependence between nodes I_2^{di} and I_1^{di}, the allocation function ensures that nodes I_2^{di} and I_1^{di} will be executed in processor elements with minimal distance between them as the schedule vector could ensure that the same nodes I_2^{di} and I_1^{di} would be executed at different moments with minimal time between executions.

The objective function is $min(\vec{X_1}, \vec{X_2})b$ subject to conditions in Equations 13:

$$\begin{array}{lrcl} 1. & \vec{X}D & \geq & 1 \\ 2. & \vec{X_1}A & = & \vec{X} \\ 3. & \vec{X_2}A & = & -\vec{X} \\ 4. & \vec{X_1} & \geq & 0 \\ 5. & \vec{X_2} & \geq & 0 \end{array} \quad (13)$$

Where $D = [\vec{d_1}, \vec{d_2}, ..., \vec{d_n}]$ is the dependence matrix.

From Equations 13,

- Condition 1 ensures data dependencies will be preserved under function X.
- Conditions 2 and 3 guarantees the convexity of the solution.
- Conditions 4 and 5 ensures the solution will have a positive direction in the convex domain.

C. Scheduling

In order to calculate the schedule function, the original linear programming problem is modified. An artificial data dependence is included in the data dependence set. Such data dependence has to be orthogonal to the allocator vector [25] to ensure that the schedule function:

- Maintains the functional behavior of the original algorithm.
- Is generated from a component independent of the allocating vector.

Two vectors, $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$ are orthogonal if their dot product is zero, where dot product is defined by Equation 14

$$a \bullet b = \sum_{i=1}^{n} a_1 b_1 + a_2 b_2 + ... + a_n b_n \quad (14)$$

The new data dependency is attached to the dependence matrix. Objective function remains the same as for calculating the allocating vector: $min(\vec{X_1}, \vec{X_2})b$ subject to conditions in Equations 15:

$$\begin{array}{lrcl} 1. & \vec{X}D & \geq & 1 \\ 2. & \vec{X_1}A & = & \vec{X} \\ 3. & \vec{X_2}A & = & -\vec{X} \\ 4. & \vec{X_1} & \geq & 0 \\ 5. & \vec{X_2} & \geq & 0 \end{array} \quad (15)$$

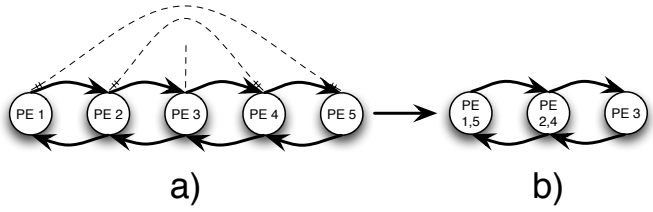

Fig. 1. Fusion approach. a) transforms into b).

Where $D = [\vec{d_1}, \vec{d_2}, ..., \vec{d_n}, (allocating\_vector)^{\perp}]$.

The combination of allocating and scheduling creates *activity holes* which are used to *compress* the resulting processor array.

D. Reducing the number of processor elements

Once a processor array has been generated by using the allocating function, the number of processor elements can be reduced. The limit for reducing the number of processor elements without decrease the throughput is the number of parallel activations. In order to maintain the structure of the data flow, a defined approach for merging processor elements is used as shown in Figure 1.

The fusion approach preserves the data flow structure by maintaining originally adjacent processors side by side.

Suppose that nodes I_x and I_y will be executed in PE_x and PE_y respectively and processor elements PE_x and PE_y will be merged in the processor element PE_z. In this case, the condition in Equation 16 guarantees that I_y will not be executed while I_x is executed.

$$\forall I_x, I_y \in IS \mid \rho(I_x) = PE_x \wedge \rho(I_y) = PE_y, \phi(I_x) \neq \phi(I_y) \quad (16)$$

Each fusion of processor elements generates a new instance of the condition in Equation 16. New conditions are included in the linear program problem in the form of data dependencies. Adding new conditions will not change the schedule function while the number of processor elements remains over the maximum parallelism.

IV. Applying the parallelization methodology to the EDA

In order to expose results of the proposed methodology, a reduced IS of 4×4 nodes is used. Because the regularity of the IS, schedule and allocating functions remains the same no matter the IS size.

A. Algorithmic representation

Recurrence equations of the proposed EDA are shown in Algorithm 1.

B. Allocation

From SURE representation, data dependence matrix is shown in Equation 17:

$$D = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 2 \\ 0 & 1 & 0 & 1 & 2 & 1 \end{pmatrix} \quad (17)$$

Algorithm 1 Edge detection algorithm (EDA)

$$
\begin{aligned}
1.\quad & p\binom{i}{j} = p_i\binom{i}{j} && \binom{i}{j} \in I_i \\
2.\quad & q\binom{i}{j} = 2 * p\binom{i-1}{j-1} && \binom{i}{j} \in I_2 \\
3.\quad & h_1\binom{i}{j} = p\binom{i-1}{j-2} + p\binom{i-1}{j} && \binom{i}{j} \in I_3 \\
4.\quad & h_2\binom{i}{j} = h_1\binom{i}{j} + q\binom{i}{j} && \binom{i}{j} \in I_4 = I_3 \\
5.\quad & v_1\binom{i}{j} = p\binom{i-2}{j-1} + p\binom{i}{j-1} && \binom{i}{j} \in I_5 \\
6.\quad & v_2\binom{i}{j} = v_1\binom{i}{j} + q\binom{i}{j} && \binom{i}{j} \in I_6 = I_5 \\
7.\quad & h_3\binom{i}{j} = h_2\binom{i-2}{j-1} + h_2\binom{i}{j-1} && \binom{i}{j} \in I_7 \\
8.\quad & h_4\binom{i}{j} = |h_3\binom{i}{j}| && \binom{i}{j} \in I_8 = I_7 \\
9.\quad & v_3\binom{i}{j} = v_2\binom{i-1}{j-2} - v_2\binom{i-1}{j} && \binom{i}{j} \in I_9 = I_7 \\
10.\quad & v_4\binom{i}{j} = |v_3\binom{i}{j}| && \binom{i}{j} \in I_{10} = I_7 \\
11.\quad & s\binom{i}{j} = h_4\binom{i}{j} + v_4\binom{i}{j} && \binom{i}{j} \in I_{11} = I_7 \\
12.\quad & p_0\binom{i-2}{j-2} = min(255, s\binom{i}{j}) && \binom{i}{j} \in I_{12} = I_7
\end{aligned}
$$

with:

$$
I_1 = \left\{ \binom{i}{j} \in \mathbb{Z}^2 | \binom{0 \le i \le N-1}{0 \le j \le M-1} \right\}, \quad I_2 = \left\{ \binom{i}{j} \in \mathbb{Z}^2 | \binom{1 \le i \le N}{1 \le j \le M} \right\}
$$

$$
I_3 = \left\{ \binom{i}{j} \in \mathbb{Z}^2 | \binom{1 \le i \le N}{2 \le j \le M-1} \right\}, \quad I_5 = \left\{ \binom{i}{j} \in \mathbb{Z}^2 | \binom{2 \le i \le N-1}{1 \le j \le M} \right\}
$$

$$
I_7 = \left\{ \binom{i}{j} \in \mathbb{Z}^2 | \binom{3 \le i \le N}{3 \le j \le M} \right\}
$$

According to the parallelization methodology presented in Section III, the data dependence conditions become:,

$$
1.\vec{X}D \begin{cases} b \ge 1 \\ a \ge 1 \\ a + b \ge 1 \\ a + 2b \ge 1 \\ 2a + b \ge 1 \end{cases} \tag{18}
$$

The allocation vector (1,1) is obtained from the linear programming approach [3], [2]. Figure 2 shows the resultant processor array by using such allocator and original data dependencies as arrows between nodes.

C. Scheduling

The next step is to include a new data dependency in the data dependence set. New data dependency has to be orthogonal to allocating vector (1,1), then the new data dependency is the vector (-1,1). Dependence matrix is extended as in Equation 19.

$$
D = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 2 & -1 \\ 0 & 1 & 0 & 1 & 2 & 1 & 1 \end{pmatrix} \tag{19}
$$

With extended dependence matrix, the new linear programming problem is solved to obtain the schedule vector (1,2).

In Figure 3, columns represent execution times $(T1, T2, ...T12)$ and rows represent iteration points I (nodes in the IS). For instance, the iteration point (1,1) will be executed in time T3 at the processor PE 4 in the reduced IS of 4×4. As a result of the projection vector in Figure 2, the first approach is to use a processor array of 9 elements. However, a more detailed analysis including Figure 3, shows that, the maximum number of parallel activations is two by using schedule vector (1,2). It means, the processor array could be reduced to two elements without decreasing the throughput.

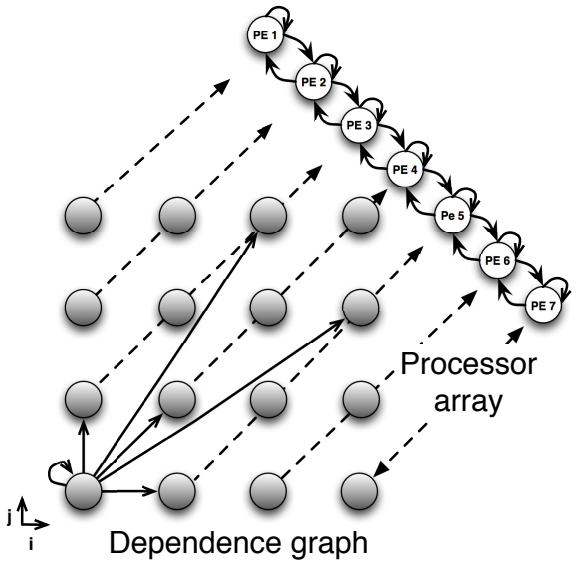

Fig. 2. Dependence Graph and processor array by using allocating vector (1,1)

i	j	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
1	1	PE 4									
1	2			PE 5							
1	3					PE 6					
1	4							PE 7			
2	1		PE 3								
2	2				PE 4						
2	3						PE 5				
2	4								PE 6		
3	1			PE 2							
3	2					PE 3					
3	3							PE 4			
3	4									PE 5	
4	1				PE 1						
4	2						PE 2				
4	3								PE 3		
4	4										PE 4

Fig. 3. Activation sequence of processor elements with schedule vector (1,2) and allocating vector (1,1).

D. Reducing the number of processor elements

In order to minimize the impact of merging processor elements in the control size, the fusion process must maintain the data flow across the execution time. In Figure 4, dotted lines indicate processor elements to be merged and columns indicate the nodes that are executed in each processor element according to the allocating vector (1,1). Data flow starts in PE 4 and propagates to extreme directions (right and left). After processor elements fusion, in Figure 5, the data flow remains constant and characteristics for processor arrays proposed in

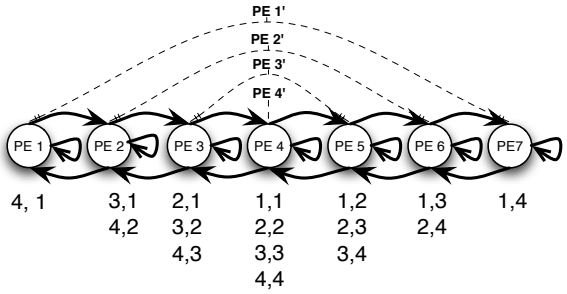

Fig. 4. Original processor array and first processor fusion.

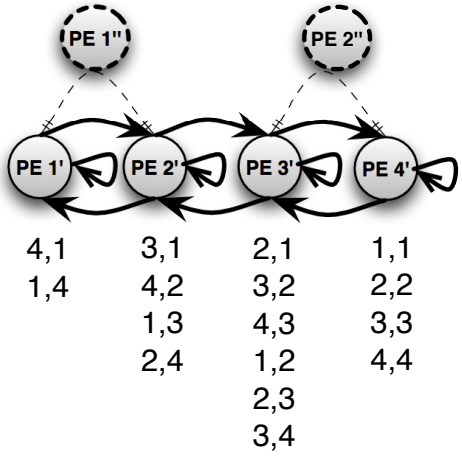

Fig. 5. Processor arrays after second and third processor fusions.

[26] are still present.
A new fusion of processor elements produces a processor array of two elements which is the minimal configuration able to maintain the throughput. A simple control system is possible, since data move between adjacent elements avoiding irregular data transmissions.
From this point, if the number of processor elements is reduced, the throughput is affected because the schedule vector is changed; however, processor array remains regular and data communications short. In the reduced IS, a last fusion generates a serial architecture where one single processor element executes all nodes I in the IS.

V. Discussion

In this paper, an extended parallelization methodology has been applied to the EDA to produce different versions of the architecture fitting different criteria.

In Figure 6, it is shown that the size of the processor array could be reduced until 25% of the original size. As the IS size grows, and after fusion process, the final processor array size approximates to 25% of the original size without decreasing the throughput.

Compressed processor array with a number of processor elements equal to the number of parallel activations is considered the first implementable version of the EDA architecture. For instance, in a problem with IS of 500×500 nodes, the original processor array requires 999 processor elements. The number of processor elements is reduced to 250 processor elements

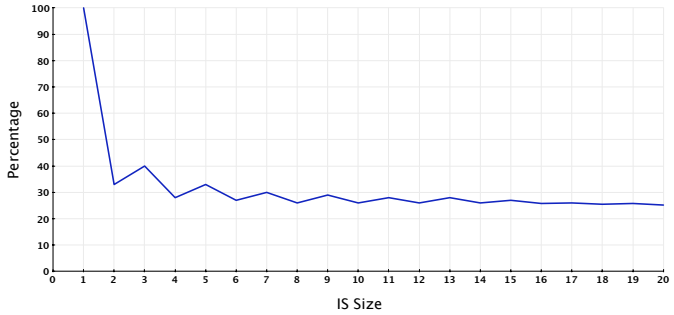

Fig. 6. Percentage of compression with different IS sizes.

Data Dependencies	0	0	1	1	1	2
	0	1	0	1	2	1
Vector (1,1)	0	1	1	2	3	3
Vector (1,2)	0	2	1	3	5	4

Fig. 7. Required memory locations per data dependency.

without affecting the execution time. Therefore, it is the first implantable version. From this point, the number of processor elements could be reduced by a half decreasing the throughput in the same proportion. Since the data flow remains unaltered, the control system remains simple.
Another contribution is related to the memory required to implement the final architecture. In Figure 7, the first two rows correspond to data dependencies in EDA. Third row presents the number of required memory locations per data dependency under schedule vector (1,1) which is the scheduler producing minimal values for w. Fourth row shows the memory locations required per data dependency under proposed schedule vector (1,2).

In total, schedule vector (1,1) requires 10 memory locations per processor element versus proposed schedule vector (1,2) which requires 15 memory locations. In the IS of 500×500 nodes, the allocation vector (0,1) which is the allocator producing minimal processor elements, produces 500 processor elements versus 250 with the proposed allocating vector (1,1). Then, the number of required memory locations with the proposed methodology is 3750 versus 5000 using alternative values.

The problem of communication in the EDA has been previously reported in [24] and [22] where data dependencies are modified to avoid large transferences of data. The main contribution of our work is that the original data dependencies are no modified as in [24]. Modifying data dependencies could change the schedule vector avoiding the optimal execution time. In this study, to modify data dependencies is not necessary because the projections approach *compress* them in $n-1$ dimensions. This reduces the distance between processor elements in the final processor array. In [22] and [24] a partitioning approach is selected based on the designer experience. By using partitions, high amounts of memory could be required to store temporal values. Partitioning and high memory requirements issues are avoided in the proposed work. It is possible since the allocating vector is obtained by an automated process allowing data naturally flow between

processor elements. Different versions of the architecture can be used as different configurations in a reconfigurable approach to meet specific requirements. From the full parallel version to the serial approach.

VI. Conclusion

In this work, a parallelization methodology applied to the EDA has been presented. EDA presents interesting challenges because the high dependence between iterations. In this paper has been shown how the proposed design methodology obtains allocating and schedule vectors and generates different versions of the required architecture. The main differences with previous works are: 1) data dependencies remain unaltered allowing optimal schedule and allocating vectors, 2) In a first stage, the number of processor elements is reduced at no cost of the execution time and 3) Required memory for implementing the processor array remains small as compared with state of the art values. The second point eliminates the main disadvantage of the projection approaches where the number of processor elements is directly related to the size of the IS thus only suitable to certain problem sizes. Results of this work can also be applied to related problems where complex data dependencies are presented.

VII. Future work

Future work includes to fit the proposed methodology under a tiling approach. Additionally, perform the necessary changes to optimize the use of memory resources directly from the linear programming problem.

References

[1] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[2] Alain Darte. Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model. *Algorithms for parallel processing, Springer Verlag*, 105:147–183, 1997.

[3] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is close to optimality. In *International conference on application specific array processors (ASAP)*, pages 37–46, 1992.

[4] Michèle Dion and Yves Robert. Towards automatic distribution. *Parallel computing*, 22(10):1373–1397, 1996.

[5] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part i, one-dimensional time. In *International journal of parallel programming*, volume 21, pages 313–347, 1992.

[6] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part ii, multidimensional time. In *International journal of parallel programming*, volume 21, pages 389–420, 1992.

[7] P. Feautrier. Towards automatic distribution. Technical report, Institut Blaise Pascal/Laboratoire MASI, 1992.

[8] Paul Feautrier. Automatic parallelization in the polytope model. Technical report 8, Laboratoire PRiSM, Université des Versailles St-Quentin, 1996.

[9] Martin Griebl, Paul Feautrier, and Armin Großlinger. Forward communication only placements and their use for parallel program construction. *languages and compilers for parallel computing*, pages 16–30, 2005.

[10] Gautam Gupta and Sanjay Rajopadhye. The z-polyhedral model. In *ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 237–248, 2007.

[11] Frank Hannig. *Scheduling techniques for high throughput loop accelerators*. PhD thesis, University of Erlangen Nuremberg, Germany, 2009.

[12] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the association for computing machinery*, 14(3):563–590, 1967.

[13] Robert H. Kuhn. Transforming algorithms for single-stage and vlsi architectures. In *In workshop on interconnection networks for parallel and distributed processing*, pages 11–19, 1980.

[14] S. Y. Kung. *VLSI array processors*. Prentice Hall, 1988.

[15] Leslie Lamport. The parallel execution of do loops. In *Communications of the ACM*, volume 17, pages 83–93, 1974.

[16] Christian Lengauer. Loop parallelization in the polytope model. *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*, 715:398–416, 1993.

[17] Lily Rui Liang and Carl G. Looney. Competitive fuzzy edge detection. *Applied Soft Computing*, 3(2):123–137, 2003.

[18] Shih-Tang Lo, Ruey-Maw Chen, Yueh-Min Huang, and Chung-Lun Wu. Multiprocessor system scheduling with precedence and resource constraints using an enhanced ant colony system. *Expert Systems with Applications*, 34(3):2071–2081, 2008.

[19] S.P.K. Nookala and Tanguy Risset. A library for z-polyhedral operations. Technical Report 1330, IRISA, 2000.

[20] Fabien Quilleré, Sanjay Vishnu Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International journal of parallel programmin*, 28(5):469–498, 2000.

[21] Radosław Rudek, Agnieszka Rudek, and Andrzej Kozik. The solution algorithms for the multiprocessor scheduling with workspan criterion. *Expert Systems with Applications*, (0), 2012.

[22] Siegel Sebastian and Merker Renate. Efficient realization of data dependencies in algorithm partitioning under resource constraints. In *Euro-Par 2006 Parallel Processing*, volume 4128, pages 1181–1191, 2006.

[23] M. Sharifi, M. Fathy, and M.T. Mahmoudi. A classified and comparative study of edge detection algorithms. In *Information Technology: Coding and Computing, 2002. Proceedings. International Conference on*, pages 117–120, 2002.

[24] S. Siegel and R. Merker. Minimum cost for channels and registers in processor arrays by avoiding redundancy. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pages 28–32, 2006.

[25] Kittitornkun Surin and Yu Hen Hu. Processor array synthesis from shift-variant deep nested do loops. *The journal of supercomputing*, 24(3):229–249, 2003.

[26] Maurice Tchuente. *Parallel Computation on regular arrays*. Manchester University Press, 1991.