Design of Processor Array Based on an Optimized Multiprojection Approach

Juan M. CAMPOS, Rene CUMPLIDO
National Institute of Astrophysics Optics and Electronics
Luis Enrique Erro No 1, Sta. Ma. Tonantzintla, 72840, Puebla, Mexico
jcampos@inaoep.mx

***Abstract*—Parallelization methodologies allow to automate the process of designing optimal processor arrays based on mathematical representations of the algorithm to be implemented. In this work, an optimized multiprojection approach based on the Polytope model is proposed as well as an automated way for getting the scheduler and the allocator vectors. Using a recurrence equations representation, three key criteria for choosing the characteristics of the final implementation are also proposed. As a case of study, the methodology is applied on a matrix-vector multiplication example. Results and relevance of the proposed methodology are finally discussed.**

***Index Terms*—Data flow computing, Parallel architectures, Parallel machines, Parallel processing, Systolic arrays.**

I. INTRODUCTION

Implementation of algorithms on hardware platforms presents multiple interesting points because the inherent capacity of hardware devices (FPGA, ASIC, etc) for parallel processing. One of the most important challenges when designing hardware architectures is to identify and to exploit the parallelism in the algorithm to be implemented. Such parallelism is limited by the data dependencies which have to be preserved in order to maintain the original behavior of the algorithm. Traditionally, design decisions are based on the experience of the designer, however, using a modeling technique it makes possible to explore the design space before the final implementation to improve some specific characteristic in design time, for instance: implementation area or throughput.

A modeling technique allows generating a processor array [1] from an algorithmic representation. Such techniques are in particular useful when the algorithm to be parallelized includes loops since loop elements are represented as points inside polyhedra [2-5]. It is possible to apply multiple transformations to the polyhedral representation in order to obtain an optimal execution time for each loop and even more, an optimal mapping between loops and processor elements to perform the original algorithm by using a design methodology.

A design methodology provides a direction vector called *scheduler vector* which indicates *when* each calculation should be performed [6-10]. Scheduler vector is a direction vector for indicating an execution order across the polyhedron as hyperplanes orthogonal to it. Aditionally to the scheduler vector, a second vector has to be proposed for indicating *where* (in which physical resource) the calculations will be executed [8-9], [11]. This second vector is called *allocator vector*; it provides a direction which is used as base in order to project points inside the polyhedron on physical processor elements generating a relationship between points and processor elements. The *allocator vector* is carefully selected since a bad decision could generate a final processor array extremely complex or to be prejudicial to the final performance. Despite the importance of the allocator vector, it is traditionally proposed by hand as in [12-13].

II. METHODOLOGY

The Polytope model [2], [14] is a mathematical tool that allows to generate efficient processor arrays for a specific algorithm. In the Polytope model, a program is modeled by a polytope, which is a finite convex set of some dimensionality with flat surfaces where every iteration in the original *loop code* is represented by one point (node) inside the polytope. The iterations set is called *Iteration Space* (IS). Figure 1 shows a characteristic polytope which represents an IS of two dimensions (i and j) including a data dependence between iterations [15-17]. Such dependence indicates that there exists some specific result in iteration 1,1 that will be required in iteration 2,1 and so on. This vector generates execution hyperplanes which are orthogonal to it and indicates that all nodes in the same hyperplane can be executed in parallel without affecting the normal behavior of the algorithm. Figure 1 shows the best possible scheduler (1,0) for this example.

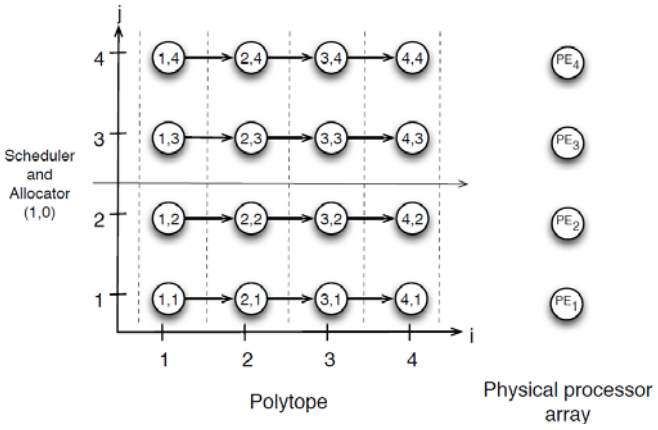

Figure 1. Polytope and data dependencies

Digital Object Identifier 10.4316/AECE.2012.04014

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

$$D = \left[\vec{d}_1, \vec{d}_2, ..., \vec{d}_i \right] \quad (1)$$

Where, $\vec{d}_i$ is the i-dependency in the dependence set. In case of Fig. 1, matrix D is composed by only one vector.
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 a linear integer program approach [6], [12], [18]. The result of the linear integer programming approach is a *linear* scheduler vector $\Lambda \in \mathbb{Z}^{1\times n}$ or equivalently expressed as a function $\phi(I)$, which assigns an execution time to every point in the *IS*
Always that it is possible to express the scheduler vector as a function $\phi(I)$ as shown in (2), the scheduler is called *linear*.

$$\phi(I) = \lfloor \Lambda I \rfloor \qquad \forall I \in IS \quad (2)$$

In (2), the linear scheduler vector $\Lambda \in \mathbb{Z}^{1\times n}$ is such that $\Lambda d_i \geqslant 1$ for all $d_i \in D$. This condition [19] is enough to ensure that all data dependencies are preserved and could be expressed as in (3).

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

where I_2^{di} and I_1^{di} are the destiny and the source nodes in *IS* of the data dependence d_i. In other words, if there exists a data dependence between nodes I_2^{di} and I_1^{di}, the condition in (3) ensures that the node I_2^{di} will be not executed before the node I_1^{di} using the scheduler Λ.
After getting the scheduler vector we have to decide where iterations will be computed (allocation) [12], [18]. The first choice for allocating iterations to physical processors is to perform a projection [20-21]. Such projection should be carefully selected since a bad choice could be unfavorable to the final throughput. A bad decision could generate a final processor array requiring more communication lines of the strictly necessaries or in the worst scenario a waste of physical resources. Allocator vector is generally proposed by hand and it is necessary to perform multiple tests in order to select the best option in terms of performance and implementation area [12-13].
In this work, the proposal is to get a first vector using the before mentioned methodology which is based on [18] and to generate a second vector by artificially including one data dependence into the original data dependence set to reformulate the linear programming problem using the modified information. This ensures an optimized allocation and congruence between scheduling and allocator vectors as well as fulfillment of the original data dependence set. Additionally, three criteria are proposed in order to select the best option as scheduler and allocator.

Once the first vector is obtained by using the methodology described in [18], the artificial data dependence is forced to be orthogonal to the first obtained vector [20] to be congruent with the flow direction proposed by the first vector. This is performed by using (4).

$$V^{\perp} = I - V^T (V \times V^T)^{-1} V \quad (4)$$

Where $V^{\perp}$ is an orthogonal vector to V, V^T is the transposed matrix of V and I is the identity matrix.
Adding the artificial data dependence to the original data dependence set ensures that:

- The original data dependence will be preserved.
- The second obtained vector is linearly independent with respect to the first vector. This means that the second vector will be congruent with the direction provided by the first vector and vice versa.

Now we have two vectors with different characteristics. Both vectors fulfill the original data dependence and are congruent with the natural data flow of the original problem. For deciding which one will be used as scheduler and which one as allocator, we propose the next considerations:

1. Communication resources.

Communication between processors is an important aspect of the hardware architecture since it could be as expensive as the processors itself. Different vectors used as allocator or scheduler will require different number of communication lines. Since processor arrays present a high regularity level, it is enough to analyze one node and its relations with the neighbors to extrapolate the findings to all elements in the array.

2. Number of processors and total execution time.

The most important premises when designing hardware architectures are the total execution time and number of processors. As in point number 1, each vector used as scheduler or allocator will produce different number of processors and different execution times.
Details for points 1 and 2 are defined from the proposed methodology next.

- *Communication resources*

  As mentioned in Section II, (3) could be used to determine if one specific data dependence is fulfilled using a scheduler vector. Additionally, (3) represents the time (w) when a datum will be transmitted from the execution point I_1 to the destiny I_2 [22] as shown in (5).

  $$\phi(I_2^{di}) - \phi(I_1^{di}) = w_i \quad (5)$$

  Where super index d_i indicates the transmitted datum fulfills the data dependence *i*. Eq. (5) is important because in case of multiple data dependencies between nodes, for example i and j, the best approach for improving the use of communication resources is to use

the same communication channel for transmitting all the required information. This is only possible if all the information have to be transmitted in different times which are defined by the scheduler vector.

As mentioned in Section II, we propose to modify the original data dependence set in order to generate a second vector as new candidate to scheduler vector. Multiple options for selecting the scheduler vector allow selecting the best option in terms of use of communication resources.

The vector that allows sending datum for each data dependence at different moments allows simplifying the communication network between processors by reusing the communication channels. Such vector could improve the use of pipeline stages in the final hardware implementation.

In case of Fig. 1, applying (5), using the vector (0,1) as scheduler vector and the only data dependence, the result is 1, which indicates that the datum will be required in the execution of the next hyper plane so, it has to be sent at the end of the actual hyperplane execution.

- *Total execution time*

Scheduling function (2) indicates when every node in the *IS* should be processed by assigning an execution time to each node. By calculating the execution time of the first and the last nodes it is possible to know the total execution time by using (6).

$$Execution\_time = \phi(I_{last}) - \phi(I_{first}) + 1 \qquad (6)$$

With reference to Fig. 1, the last hyperplane will be executed at time 4 and the first hyperplane will be executed at time 1, using (6) it is possible to calculate the total execution time under the proposed scheduler in 4 time units.

- *Number of processor elements*

The size of the processor array is a key point to be considered since it has a direct relation with power consumption. For mapping some nodes of the *IS* point to the corresponding processor element under the allocator vector it is possible to use (6). However, in this case results are interpreted as processor IDs instead of execution times. In case of Fig. 1, the higher index for an PE is 4 and the lowest index for a PE is 1 then, the total number of required processors according to (6) is 4. It is important to note that the scheduler and the allocator could be different.

III. Example. Matrix-vector multiplication

In order to show the proposed methodology, we present an example using the matrix-vector multiplication. For representing the algorithm we propose to use a set of recurrence equations which allows a cleaner representation avoiding the overhead of a C representation as in [23][24]. Additionally, a C representation imposes an *a priori* execution order which is avoided by using the recurrence equations representation. Equations set (7) shows the proposed representation using a set of recurrence equations [25] corresponding to the matrix-vector multiplication example.

$$\begin{aligned}
&1.\ a(i,j) = A_{i,j} && 1 \le i,j \le N;\\
&2.\ b(i,j) = \begin{cases} b_j & i = 1, 1 \le j \le N;\\ b(i-1,j) & 1 < i \le N, 1 \le j \le N; \end{cases}\\
&3.\ z(i,j) = a(i,j) \bullet b(i,j) && 1 \le i,j \le N;\\
&4.\ c(i,j) = c(i,j-1) + z(i,j) && 1 \le i,j \le N;\\
&5.\ C(i) = c(i,j) && j = N;
\end{aligned} \qquad (7)$$

Where *a(i, j)* is the input variable for values of the matrix *A* and *b(i, j)* is used as the input variable for values of the vector *B* and as temporal variable to propagate the values of the vector *B*. The temporal variable *z(i, j)* stores the result of intermediate multiplications, *c(i, j)* is used as accumulator and finally, *C(i)* is the vector result of the multiplication. Fig. 2 shows how all of these variables are related using *N=4*. Fig. 2 shows the data dependencies as connections between nodes too.

Equations set (7) includes two data dependencies, the first in line 2 and the second in line 4 which correspond to the two column components of the dependence matrix *D*, shown in (8).

$$D = \begin{bmatrix} \vec{d}_x & \vec{d}_y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad (8)$$

From (7) we can get the dimensionality of 2 (*i* and *j*) of the *IS*. Finally the size of each dimension corresponds to the interval where every dimension (*i* and *j*) is defined, in this case *N*.

Linear programming problem requires conditions for the objective function. In this case the only condition is to respect data dependencies [3]. Such condition is represented by (3) which indicate that for any possible solution to the linear programming problem is not possible to execute calculations in node (1,2) before or even in parallel with calculations in node (1,1) if there is a data dependence from node (1,1) to node (1,2).

Solving the linear programming approach on the dependence matrix and the *IS* information we get the first proposed vector (1, 1). As previously mentioned, the next step is to include one new data dependence to the original data dependence set. The new data dependence has to be orthogonal to the first obtained vector. Eq. (2) is used to get the (-1, 1) vector which is included in the original data dependence set. Basically we transform the original *IS* shown in Fig. 2 into the shown in Fig. 3.

The linear programming approach is solved on the new data dependence set to get the vector (2, 1). Now, we have two different vectors (1, 1) and (2, 1). Both vectors fulfill the original data dependencies and are congruent with the data flow. This means that any of both vectors could be used as scheduler vector or as allocator vector without affecting the normal behavior of the algorithm.

The second vector was generated using an extra component orthogonal to the first vector and finally they present different characteristics. We propose that the final decision about which one will be used as scheduler and which one as allocator depends on the following three aspects: Communication resources, number of processor elements and total execution time that are discussed next.

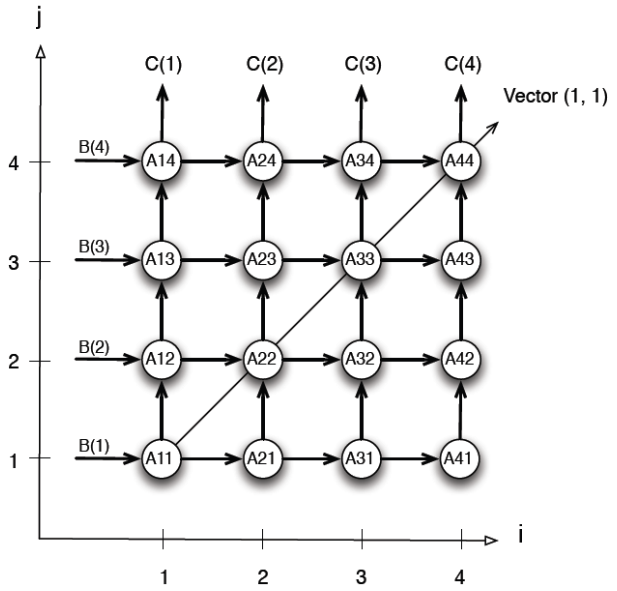

Figure 2. Original polytope of the matrix-vector multiplication.

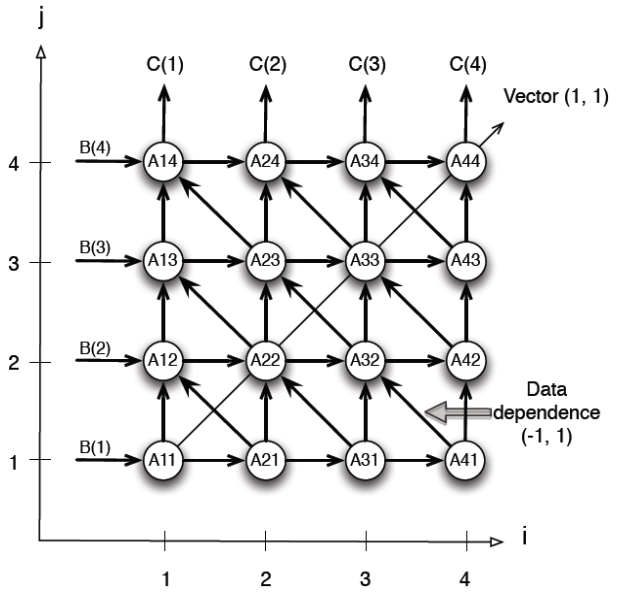

Figure 3. Polytope of the matrix-vector example with modified data dependencies.

- *Communication resources*

According to (3), every data dependence is evaluated using both vectors. Results are shown in Table I.

TABLE I. SENDING TIME FOR DATA DEPENDENCIES ON DIFFERENT SCHEDULER VECTORS

Data dependencies	Sending time using vector (1,1)	Sending time using vector (2,1)
1,0	1	2
0,1	1	1

From Table I, values in column 2 indicate that using the vector (1, 1) as scheduler requires sending both data at the same time however, from column 2 using the vector (2, 1) as scheduler vector, data could be sent at different moments. At this point, it is better to select the vector (2, 1) as scheduler since such vector allows saving communication resources by reusing communication channels or even implementing pipeline stages. However, it is important to consider other aspects such as area and execution time which are revised next.

- *Number of processor elements*

Area resources are, in general, a key target when implementing some algorithms in hardware platforms because the available resources are limited. Fig. 4 shows the number of required processors by using as allocator the vector (1, 1) and the vector (2, 1) on different *IS* sizes. Using the vector (1, 1) as allocator requires less processors by a factor close to 1/3. According to Fig. 4 this factor is a constant, independently of the *IS* size in this specific problem.

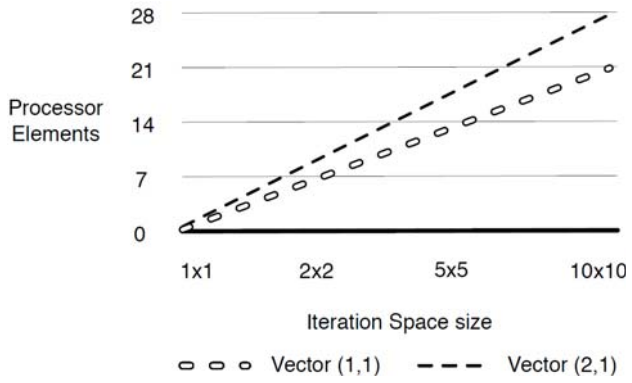

Figure 4. Polytope of the matrix-vector example with modified data dependencies.

- *Total execution time*

As mentioned in Section II, (2) provides a practical way for knowing the execution time of each node in the *IS* as well as the number of required physical processors. Since the *IS* of the proposed example is square, it is possible to reinterpret Fig. 4 as a time graphic which means that using vector (1, 1) as scheduler vector implies a faster processing time than using vector (2, 1) as scheduler vector. In this case, the difference between total execution times between both vectors is the same constant factor close to 1/3.

Fig. 5 shows the final projection vector (1,1) and the scheduler vector (2,1). In this case, a compact implementation is preferred even at the expense of the required processing time which as before mentioned is greater by a constant factor close to 1/3. Using the vector (1, 1) as scheduler vector, the number of processor elements is minimal. Fig. 5 shows a graphical representation of the scheduler vector and the generated hyperplanes. As mentioned in Section I, all the nodes in the same hyperplane are executed in parallel. The time used in each hyperplane is the required time to complete the calculations in a single node.

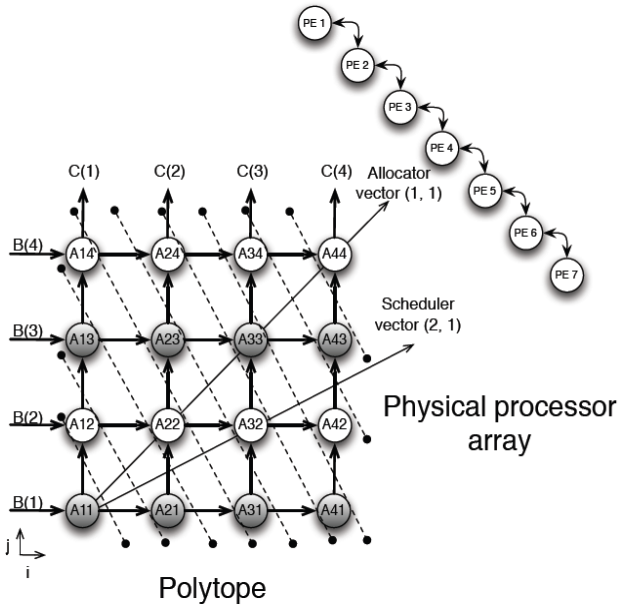

Figure 5. Final processor array for matrix-vector multiplication example with allocator vector (1,1)

If the interest is focused on reducing the execution time, it is possible to select the vector (2,1) as projection vector and the vector (1,1) as scheduler vector. Such approach provides a higher throughput at the expense of increased number of physical processors. Final processor array is shown in Fig. 6 Here, execution time is the smallest possible execution time limited only by data dependences however the communication is more complex when compared with using the vector (2, 1) as scheduler vector. More complex communication network requires a more complex control system.

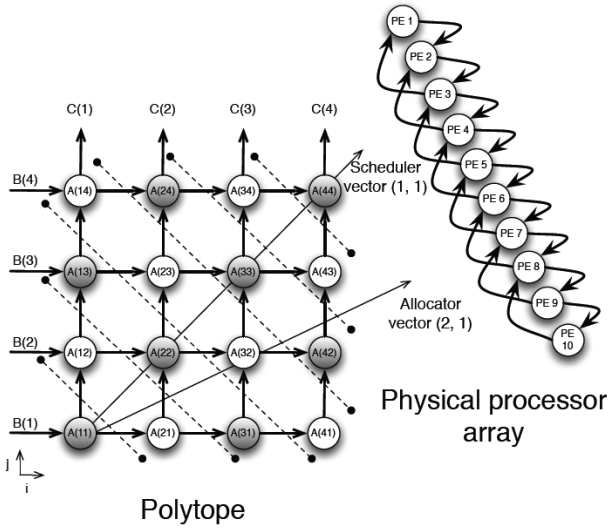

Figure 6. Final processor array for matrix-vector multiplication example with allocator vector (2,1)

As an example, Table II indicates in the second column, the number of processor elements and, in the third column the processing time expressed in hyperplanes, all of this calculated on an *IS* with N=100. Values are calculated using (6). Table II presents symmetry since the data dependences in the *IS* presents the same characteristic.

TABLE II. PROCESSOR ELEMENTS AND PROCESSING TIME UNDER DIFFERENT SCHEDULER VECTORS. N=100

Scheduler Vector	Number of processor elements	Processing time
1,1	199	299
2,1	299	199

No matter the case, the proposed methodology prevents the manual selection of the projection vector as in [12][20] generating an optimized second vector which is congruent with the data flow. Having multiple choices for selecting the scheduler vector provides flexibility in defining the best projection vectors under certain criteria. Proposed aspects for selecting scheduler and allocator vectors are easily evaluated using (2). Methodology starts with a recurrence equations representation which avoids the overhead and the *a priori* order of a C representation.

IV. CONCLUSIONS

In this work, a parallelization methodology based on a multiprojection approach is proposed. The methodology provides two vectors and three criteria for deciding which vector is used as scheduler vector and which is used as allocator vector. The allocator vector is obtained according to an optimization process with a linear programming approach instead of using manual selection as in previous works. In this work, the original algorithm is represented as recurrence equations which avoids the overhead of using a C representation. The entire process could be fully automated since user intervention is not required. In order to demonstrate the proposed methodology, in the second part of the paper, the methodology was applied on the matrix vector multiplication example. Results on the proposed example demonstrate that the proposed methodology is successfully implemented allowing flexibility in design time and optimizing the execution time or the number of processor elements in the processor array.

REFERENCES

[1] S. Y. Kung. “VLSI Array Processors”. Printice Hall, Englewood Cliffs, New Jersey, 1988.

[2] Christian Lengauer. “Loop Parallelization in the Polytope Model”. In Eike Best, editor, Proceedings of the 4th International Conference on Concurrency Theory (CONCUR), volume 715 of Lecture Notes in Computer Science (LNCS), pages 398–416, Hildesheim, Germany, August 1993.

[3] Alain Darte. “Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model”. In M. H. Heath, A. Ranade, and R. S. Schreiber, editors, Algorithms for Parallel Processing, volume 105 of IMA. Volumes in Mathematics and its Applications, pages 147–183. Springer Verlag, 1998.

[4] S.P.K. Nookala and Tanguy Risset. “A library for Z-polyhedral operations”. Technical Report PI 1330, IRISA, Rennes, France, 2000.

[5] Gautam Gupta and Sanjay Rajopadhye. “The z-polyhedral model”. In ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pages 237–248, 2007.

[6] P. Feautrier. “Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time”. International Journal of Parallel Programming, 21(5):313–348, 1992. Available: http://dx.doi.org/10.1007/BF01407835.

[7] P. Feautrier. “Some efficient solutions to the affine scheduling problem: Part II, multidimensional time”. International Journal of Parallel Programming, 21(6):389–420, 1992. Available: http://dx.doi.org/10.1007/BF01379404.

[8] P. Feautrier. “Toward automatic distribution”. Parallel Processing Letters, 4:233–244, 1994. Available: http://dx.doi.org/10.1142/S0129626494000235.

[9] Michele Dion and Yves Robert. “Mapping affine loop nests”. Parallel Computing, 22(10):1373–1397, 1996. Available: http://dx.doi.org/10.1016/S0167-8191(96)00049-X.

[10] Martin Griebl. “Automatic Parallelization of Loop Programs for Distributed Memory Architectures”. University of Passau, 2004. Habilitation thesis.

[11] Martin Griebl, Paul Feautrier, and Armin Großlinger. “Forward communication only placements and their use for parallel program construction”. In Languages and Compilers for Parallel Computing, pages 16–30. Springer- Verlag, 2005. Available: http://dx.doi.org/10.1007/11596110_2.

[12] Frank Hannig. “Scheduling Techniques for High Throughput Loop Accelerators”. Dissertation, University of Erlangen Nuremberg, Germany, August 2009. Verlag Dr. Hut, Munich, Germany.

[13] Fabien Quilleré, Sanjay Vishnu Rajopadhye, and Doran Wilde. “Generation of Efficient Nested Loops from Polyhedra”. International Journal of Parallel Programming, 28(5):469–498, 2000. Available: http://dx.doi.org/10.1023/A:1007554627716.

[14] Paul Feautrier. “Automatic Parallelization in the Polytope Model”. Technical Report 8, Laboratoire PRiSM, Université des Versailles St- Quentin en Yvelines, 45, avenue des États-Unis, 78035 Versailles Cedex, France, June 1996.

[15] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. “Facilitating the search for compositions of program transformations”. In ACM International conference on Supercomputing, pages 151–160, June 2005.

[16] Nicolas Vasilache, Cedric Bastoul, Sylvain Girbal, and Albert Cohen. “Violated dependence analysis”. In ACM International conference on Supercomputing, June 2006.

[17] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. “Iterative optimization in the polyhedral model: Part I, one-dimensional time”. In International symposium on Code Generation and Optimization, March 2007. [Online]. Available: http://dx.doi.org/10.1109/CGO.2007.21.

[18] Alain Darte, Leonid Khachiyan, and Yves Robert. “Linear Scheduling is Close to Optimality”. In Proceedings of the International Conference on Application Specific Array Processors (ASAP), pages 37–46, Berkeley, CA, USA, August 1992. [Online]. Available: http://dx.doi.org/10.1109/ASAP.1992.218583.

[19] Leslie Lamport. “The parallel execution of do loops”. Communications of the ACM, 17(2):83–93, 1974. [Online]. Available: http://dx.doi.org/10.1145/360827.360844.

[20] Kittitornkun Surin, Yu Hen Hu. “Processor Array Synthesis from Shift-Variant Deep Nested Do Loops”. The Journal of Supercomputing, 24(3):229-249, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1022028729196.

[21] Robert H. Kuhn. “Transforming Algorithms for Single-Stage and VLSI Architectures”. In Workshop on Interconnection Networks for Parallel and Distributed Processing, pages 11–19, West Layfaette, IN, USA, April 1980.

[22] Michael Wolfe. “High Performance Compilers for Parallel Computing”. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. 2007. “Automatic Correction of Loop Transformations”. In Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07). IEEE Computer Society, Washington, DC, USA.

[24] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. “Automatic mapping of nested loops to FPGAS”. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07). ACM, New York, NY, USA, 101-111.

[25] 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. [Online]. Available: http://dx.doi.org/10.1145/321406.321418.