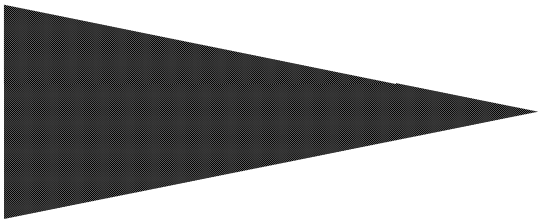
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1335



USING KNAPSACK TECHNIQUE TO PLACE LINEAR ARRAYS ON FPGA

ERWAN FABIANI AND DOMINIQUE LAVENIER

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Using knapsack technique to place linear arrays on FPGA

Erwan Fabiani and Dominique Lavenier

Thème 1 — Réseaux et systèmes
Projet COSI

Publication interne n1335 — Juin 2000 — 16 pages

Abstract: This report presents a methodology for mapping linear processor arrays onto FPGA components. This methodology is based on the analogy of this mapping problem with the knapsack problem : for a given knapsack with a specific weight capacity, and for a set of objects characterized by their weight and their profit, find the best subset of objects to put in the knapsack for optimizing the profit without exceeding its weight capacity. We show how a standard knapsack problem resolution is adapted to define a placement that take advantage of regularity and locality properties of linear array structures. This placement allow vendor tools to skip this phase and produce fast and optimized routing.

Key-words: regular arrays, fast placement, FPGA

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Utilisation du problème de sac à dos pour le placement de réseaux réguliers sur circuits FPGA

Résumé : Ce rapport présente une méthode pour placer des réseaux réguliers de processeurs sur des composants FPGA. Cette méthode est basée sur l'analogie de ce problème de placement avec le problème du sac à dos : étant donné un sac à dos d'une certaine capacité d'emport de poids et étant donné un ensemble d'objets caractérisés par leur poids et leur profit, trouver l'ensemble d'objets à mettre dans le sac à dos qui maximise le profit tout en n'excédant pas la capacité du sac à dos. Nous montrons comment une méthode standard de résolution du problème du sac à dos est adaptée pour produire un placement qui tire avantage des propriétés de régularité et de localité des réseaux réguliers. Ce placement permet aux outils standards de passer la phase de placement et de réduire la phase de routage.

Mots clés : réseaux réguliers, placement rapide, FPGA

1 Introduction

In many compute intensive applications such as image or signal processing, time is mostly spent in executing loops. Speeding-up these applications, for example under real-time constraints, leads to hardware implementations which directly benefit from the inherent loop parallelism. The resulting architecture is a regular array, often a systolic array, made of simple processing elements dedicated to efficiently performing the body of the inner loops [1]. The structure can either be 1 or 2-dimensional array, but in the following we will restrict to linear arrays only. This restriction is justified since it is possible to transform a 2-dimensional array into a 1-dimensional array by serialization. Moreover, the implementation of a 2-dimensional array without serialization could result in an array throughput which will exceed available input/output capabilities.

Implementing such nested loops onto FPGA components presents many advantages. First, the regular nature of FPGA component – an array of small bit-processing elements – matches perfectly the architecture we focus on: a replication of identical regularly interconnected processing elements. Second, the best uses of FPGA boards (from a performance point of view) have been demonstrated on many compute intensive applications, as illustrated by the numerous applications implemented on the PAM boards [2]. Third, new advanced microprocessor architectures tend to incorporate reconfigurable resources in their data-path. Parallelizing loops on these specific areas is a very attractive way to efficiently exploit reconfigurable computing.

We advocate that if, today, implementing high performance architectures on reconfigurable platforms is technically feasible, the main restriction comes from the programming tools. We still code these "programmable" devices as the first programming pioneers did when they were using assembly languages for programming early computers. Efficient implementations are achieved with a very good knowledge of the FPGA component structure together with a long experience of how to handle the entire tool outfit presently required to create a design. Without putting down the research of new reconfigurable computers, which must continue to be imaginative, we believe that their success is also tied to the ability to provide high level programming tools. Our work goes in this direction.

More precisely, the reconfigurable computing research project of the COSI team at IRISA, aims to automate the hardware parallelization of nested loops onto FPGA boards. There are three major steps as described in [3]:

- **Parallelization:** This step consists in deriving regular array architectures (systolic as well as semi-systolic) from loop specifications or equivalent formal description such as systems of *affine recurrence equations*. This model supports a powerful theory of space-time transformation methods which are now also used for automatic parallelization. The ALPHA language, developed at IRISA allows the programmer to explore transformations needed for systematic derivation of regular arrays and for automatic parallelization [4] [5].
- **Partitioning:** Since the available reconfigurable resources have physical limits, and may not support the entire array, transformation of the architecture is required. It con-

sists in splitting the array into sub-arrays or clustering groups of processing elements. The automating of this task is still ongoing research and is not yet fully resolved.

- **Physical Mapping:** This last step maps the architecture on the reconfigurable support. From a RTL description (provided by the previous stages), one must find the best mapping both in term of speed performance and area occupation. This is actually a very time-consuming step which tends to become longer as the FPGA components grow in complexity.

The work presented in this report deals with the last stage. It focuses on reducing the place-and-route process involved in the physical mapping task by taking advantage of the regular nature of the of the array we want to map.

The next section exposes first the regular place-and-route foundation. Section 3 presents related work. Section 4 explains our strategy for mapping linear array onto FPGA. Section 5 concludes.

2 Regular Place-and-Route Foundation

The place-and-route process is generally performed by the vendor tools associated with the FPGA component family because access to low level details are proprietary. The input is often a flat RTL architecture (i.e. without hierarchical structure), and the goal is to find the best match between the FPGA component structure and the circuit to implement, knowing that the routing channels are resource limited. In other words, this induces distributing first the boolean equations and registers among the FPGA bit-processing elements (referred in the following as LUT since a common implementation is based on Look-Up Table) in such a way that the connections are minimized. Then, the routing process can take place. Generally, better the placement, faster the routing.

However these two steps are very time consuming, especially with the larger FPGA components. This is mainly due to the algorithmic techniques (such as simulated annealing) used for finding reasonable solutions. The advantage of these techniques are their generality: they provide relatively good solutions whatever the structure of the designs. In our case, as we try to shift towards software compilation requirement, the major drawback is definitely the computation time.

One way to limit this time is to provide the best placement as possible to optimize the routing phase. The methodology we developed for mapping regular arrays onto FPGA components is mainly based on this idea. Our thesis is that placing an array of processing elements according to its regular and locality properties brings three major improvements over usual place-and-route techniques:

1. the placement time is drastically reduced;
2. the routing time is optimized;
3. the frequency is increased;

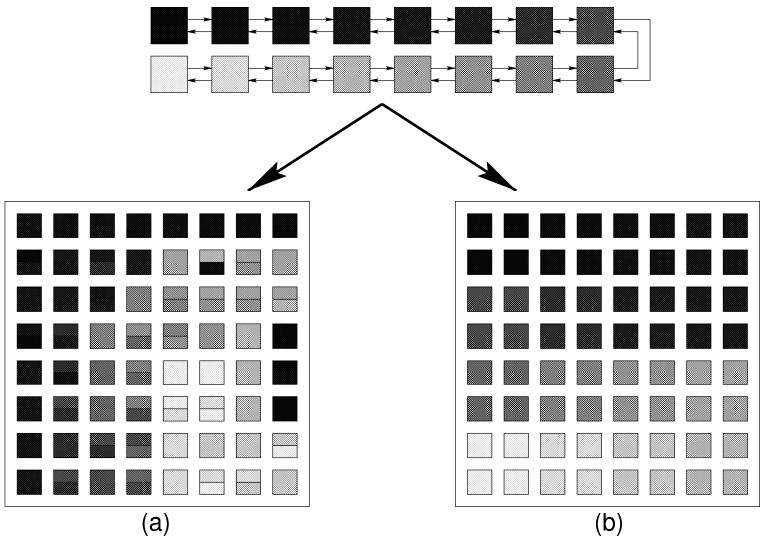


Figure 1: Comparison of 2 physical mappings of a linear array *(top of the figure)* : *(a)* without pre-defined placement; *(b)* with pre-defined placement. A processing element fits into 8 LUTs (4 logic blocks). In *(a)*, the intrinsic regularity of the array is partially respected but some processing elements are split into distant LUT, and logically adjacent processing elements are not physically close. In *(b)* a processing element is clustered into 8 adjacent LUTs, and the closeness of logically adjacent processing elements is scrupulously respected.

Our placement strategy for taking advantages of these improvements is based on the following rules:

1. Signals which belong to a same processor have their sources placed in a same restricted area. This implies:
   - the reduction of the placement time: the possibilities for distributing the boolean equations generating these signals over an enclosed area are limited;
   - the reduction of the routing time: the possibilities for routing signals in a restricted area are limited;
   - the reduction of the delay: the connections inside a limited area are short.
2. Identical processors have identical placement: the placement focuses only on one processing element and is replicated over the FPGA component. The time is thus independent of the number of processing elements.
3. Neighboring processing elements are close to each other. The expected benefits are:
   - a reduction of the placement time: the possibilities for placing the processing elements are actually very limited;
   - a reduction of the routing time: again, the possibilities for routing the connections between processing elements connections are limited;
   - a reduction of the delay: short connections between processing elements.

Figure 1 illustrates our placement strategy: in addition to reflecting the regularity of the linear array architecture, finding the best 8-LUT arrangement and replicate it over the FPGA array is obviously faster than blindly seeking a global mapping.

A few experiments have been carried out to validate this thesis, and are summarized in the figures 2 and 3. Basically, we compare the time to place-and-route a design with and without placement directives. The expected frequency (given by the place-and-route tool) is also reported.

In figure 2, the comparison unit is given by the time to place-and-route an unplaced design. In figure 3, we compare the speed-up induced by designs with placement directives against designs without placement directives. This comparison is done for the placement step runtime, the routing step runtime, the place-and-route runtime and the clock frequency.

The designs are very regular structures and are taken from real applications:

- **Lyon** is a linear systolic implementation of the Lyon's bit-serial multiplier [6]. The length of the array is equal to the data word length.
- **Conv** is a systolic implementation of a 16-bit pseudo-convolution in which the multiplier has been replaced by a logical AND.

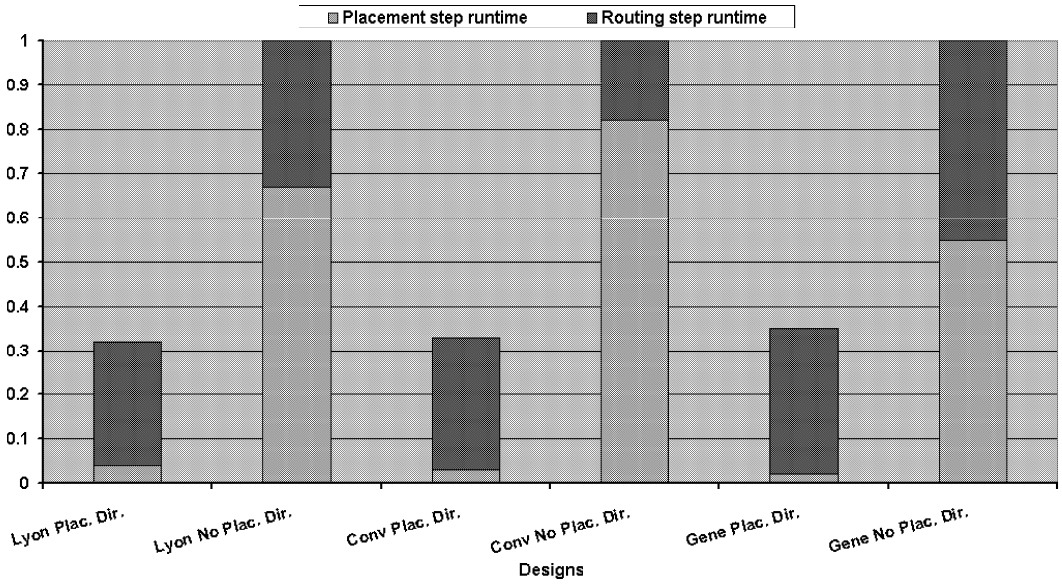


Figure 2: Average runtime comparison to place-and-route a design with (Plac. Dir.) or without placement directives (No Plac. Dir.), on a normalized unit scale

- **Gene** implements the mutation part of a systolic genetic algorithm [7]. An elementary 8-bit processor is composed of a shift register, an accumulator, a comparator, and three bank registers.

The designs *Lyon, Conv, Gene* are implemented on a XC4020 Xilinx [8], using the PPR Xilinx place-and-route tool. Placement is achieved using the constraint directives provided by PPR.

Actually, the values given in the above table are average values: for each design several place-and-route runs have been performed with different options available on the router.

In these examples, we observe that the placement phase is more time consuming than the routing phase, and shortening this step results in a significant speed-up, even if the routing phase, in some cases (cf. *Conv*), increases.

Of course, these experimental results must not be considered as definitive values, but rather as tendencies confirming our intuitive idea. They just indicate that a pre-placement step is interesting in the case of regular architectures, and, further, that it does not lead to degraded clock frequency.

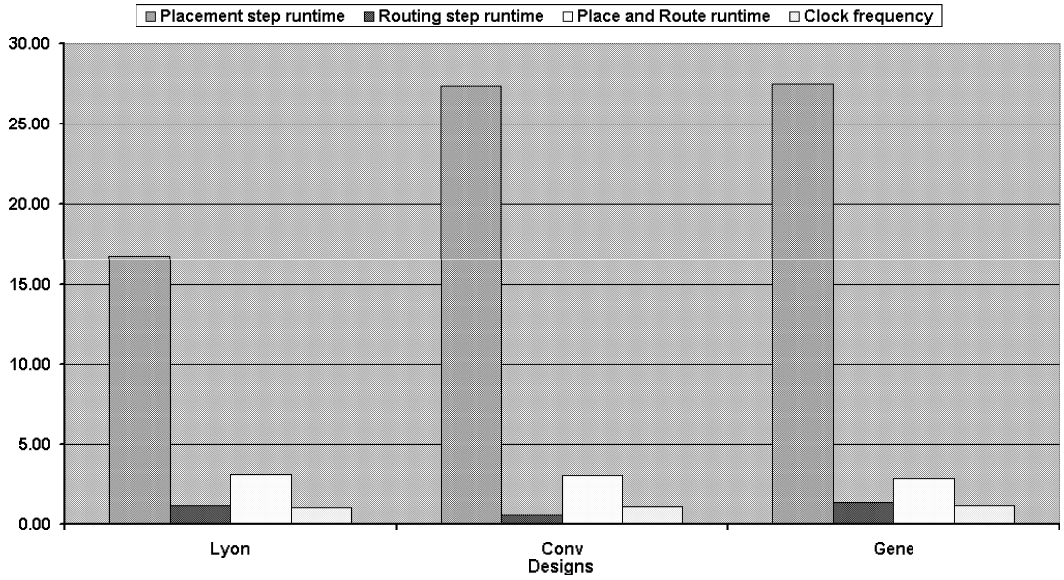


Figure 3: Comparison of runtime speed-up and clock frequency speed up for a design with placement directives

3 Related Work

The *Array Mapper* [9] automatically place 2-dimensional processor arrays. It allocate a non rectangular tile to each processor. This tile is identical for all processors and is selected so that the array of processor tile covers the logic bloc array without gaps and overlaps. These tiles can be arranged in a rectangular or hexagonal grid which respectively results in having 4 or 6 neighbors for each processor.

The *Array Mapper* first find the shape of the tile : given the netlist of processors, the specification of links between processors, the grid type and various constraints, it find an initial shape using mincut algorithm. Then random distortions of this shape are performed to improve the critical path. Once the tile shape is fixed, a simulated annealing procedure place the logic blocks of the processor into the tile.

However *Array Mapper* does not take the physical mapping runtime into account, is dedicated to small size processors (about ten logic gates) and is targeted to fine grain FPGAs.

Others tools take physical mapping runtime into account, but consider the automatic placement of modules in spite of linear arrays. However, since they try to deduce the placement from the design structure and datapath, their results are interesting regarding to our work.

SDI

SDI [10] aims to keep regular datapath structure during the physical mapping step in order to reduce compilation runtime and to increase clock frequency. The designs targeted by *SDI* are very regular : they are modeled as a regular datapath of modules and each module is compound of basic slices. In SDI, the design implementation is globally done in three steps:

- A set of possible physical implementation is generated for each module
- The choice of each module implementation and their linear placement is done simultaneously using a genetic algorithm. Criterions used during this step are the uniformity of module implementation, the connection length between modules and the possibilities of adjacent module compaction.
- A compaction of adjacent modules is tried : standard synthesis algorithm is locally applied to adjacent module slices, keeping the module sliced structure.

The placement found by *SDI* is then expressed with placement constraints to be used by vendor tools. Results show a 2.47 average speed-up of the place and route runtime and a 15 % average increase of clock frequency.

GAMMA

GAMMA [11] uses a tree covering method to produce fast datapath module mapping. Designs are described as a dataflow graph in which each nod is an operator. *GAMMA* relies on a module library and each module can implement one or several operators. Thus *GAMMA* tries to find the best covering of operators using these modules and place them. *GAMMA* operates in four steps :

- The design dataflow graph is converted into a DAG which is split into trees.
- Each tree is covered in topological order using module library. The linear order of the chosen modules is done simultaneously : it is determined by the tree covering. For modules of the same order, all possible placements of these modules are tested and the one that minimizes the net lengths is chosen.
- Local optimization is performed at boundary of adjacent trees.
- Each selected module is generated given datapath width, constants inputs, etc.

As *SDI*, the output of *GAMMA* is a netlist file containing mapping and placement constraints which is routed by vendor tools. Compared to a full place and route performed by vendor tools, the place and route runtime is speeded up by 2.06 on average using *GAMMA*.

Bipartitioning

Another method of fast placement that relies on design structure is the bipartitioning based floorplanner [12] . The input designs are based on a set of predefined macros. These macros have a fixed area and a fixed or flexible shape. The floorplanner determine a physical placement for fixed shape macro. For flexible shape macro, it find the shape dimension, the internal placement and the physical placement. The floorplanner operates in three steps :

- The set of macros is successively bi-partitioned in subsets, keeping highly connected macros in a same subset. In the same time, the area slice is divided in the same way, with alternated horizontal or vertical cuts, in order to have one area slice associated with each macro subset. The successive bipartitioning is terminated when the cardinality of each subset is inferior to a fixed number.
- For each area slice, an exhaustive search of the best relative placement of its macros is done. Since the number of macros by subset is small, this exhaustive search do not cost too much time. Then the fixed shape macros are placed. For each flexible shape macro, adjacent columns of logic blocks are allocated on the area slice to contain the macro CLBs. These CLBs are placed in these columns using simulated annealing.
- After the floorplanning there could be unused CLBs. The step of compaction eliminate unused rows and columns of CLBs from the floorplan.

The bipartitioning method obtain a 2.94 average speed-up of the place-and-route runtime. The FPGA area used is increased or decreased according to the design (from +33 % to -29 %), but the clock frequency is similar.

4 Regular Place-and-Route Strategy

4.1 The FPGA Regular Array Placer (FRAP)

Figure 4 details the place-and-route environment for mapping regular arrays onto FPGA components. The input is a RTL description **without** placement directives.

The regular placement is performed with the FRAP tool and acts in three steps:

1. All possible shapes for a processing element are generated by combining all shapes of its sub-components.
2. A full *snake* placement of the linear array is determined using the processing element shapes previously computed.
3. The final placement of the processing elements are performed according to their shapes.

Steps 1 and 3 deal with processing element placement. We consider those elements rather small, that is a few operators essentially coming from a library, and that finding

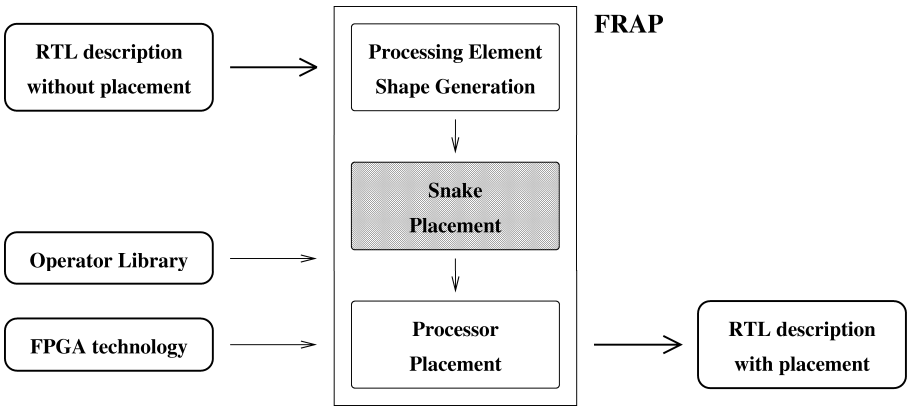


Figure 4: The FPGA Regular Array Placer (FRAP): the RTL description given as an input is processed and output with placement directive annotations.

a good placement is a fast and non critical process. On the other hand, step 2 is more interesting as explained in the next sections.

The FRAP output is also a RTL description, possibly modified but equivalent to the input one, **with** placement directives. From this description an EDIF file is generated and input to the vendor place-and-route tools. Since the placement is fully specified, the computation time is reduced to roughly the time for routing the FPGA component.

4.2 The Snake Placement Strategy

The problem is to place a linear array on a 2-dimensional FPGA structure. The only way to keep two neighbor processing elements close to each other is to implement a snake-like arrangement of the array. The determination of the snake-like arrangement proceeds in two phases: (1) divide the FPGA area in sub-areas that we call *convenient areas*, and (2) for each convenient area, place a maximum number of processing element in a snake-like fashion.

Convenient Area Partitioning:

This step is required because of the physical I/O constraints of the reconfigurable support. The first and the last processing element of an array cannot be located anywhere. They must be implemented near some specific physical I/O connectors.

A convenient area is defined as a rectangular area in which the locations of the first and the last processing elements are situated in two different corners. If, initially, the full FPGA area is not a convenient area (that is, if the I/O constraints impose that either the first or the last processing element is not located in a corner), then this area must be divided into convenient sub-areas.

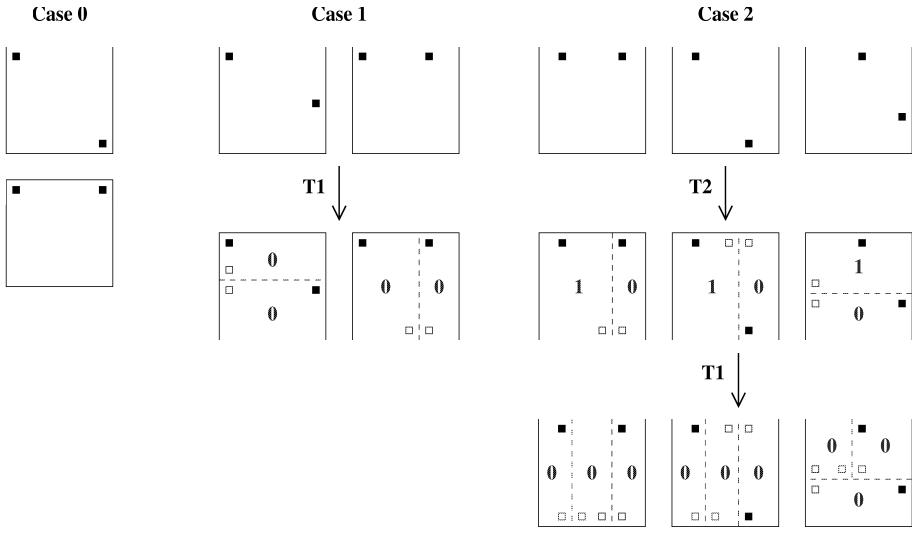


Figure 5: Partitioning into convenient area. Three cases must be considered according to the boundary processing element locations. An operation (of type T1 or T2) divides the area into two sub-areas in such a way that at least one boundary processing element is located in a corner of a new area.

The convenient areas are determined by analyzing the location of the first and the last processing elements (boundary processing elements) and applying a corresponding splitting operation. Three cases may occur:

- **case 0:** Boundary processing elements are in two different corners. This is a convenient area. No further processing.
- **case 1:** Only one boundary processing element is in a corner. Apply operation T1.
- **case 2:** No boundary processing elements located in a corner. Apply operation T2.

An operation (of type T1 or T2) consists in separating the area in two new sub-areas with at least one boundary processing element located in a corner in one of the two new sub-areas. To each new created sub-area a fictive boundary processing element is also created. Note that it is always in a corner. The process is recursively applied on the sub-areas which are not convenient areas. Figure 5 gives some representative but not exhaustive examples of the process. As the operations applied to case 2 reduces the new sub-areas to case 1 or case 0, and operation applied to case 1 reduces the new sub-areas to case 0, the partitioning into convenient areas is at most performed in two iterations.

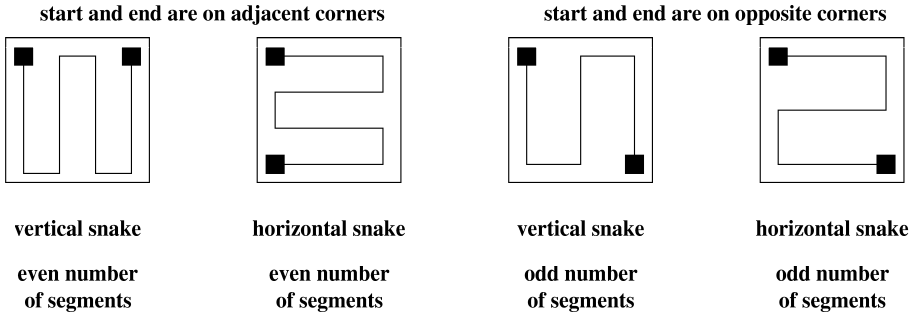


Figure 6: Determination of the snake orientation and the number parity of the segments according to the location of the boundary processing elements.

Snake Placement:

First, the orientation and the segment number parity of the snake are determined according to the location of the boundary processing elements (see figure 6). A segment is a line (or column) of processing elements. Thus, a snake is specified by two parameters: its orientation (vertical or horizontal) and the segment number parity. The problem is now to fit a maximum number of processors in the convenient area according to these two parameters. In order to get as much as flexibility as possible, and also to avoid wasting space, we do not insist that two segments belonging to the same snake are composed of processing elements with identical shape (since we have a collection of possible shapes for the processing elements).

This is typically an optimizing problem and we solve it using the knapsack metaphor: for a given knapsack with a specific weight capacity, and for a set of objects characterized by their weight and their profit, find the best subset of objects to put in the knapsack for optimizing the profit without exceeding its weight capacity [13].

In our case, the objects are represented by the segments, the weight of an object is the height of a segment (the dimension perpendicular to the data-flow), the profit corresponds to the number of processors in one segment, and the capacity of the knapsack is the height of the convenient area (see figure 7.b). The diversity of the objects (the segments) comes from the ability for a processing elements to have several shapes.

The Knapsack problem is usually solved by dynamic programming where a function $f(j,k)$ is recursively computed [14]. The variable j represents the capacity (the height of the convenient area) and k the number of objects (segments). Actually, we tailored the general formula to reject some solutions (such as a wrong parity of segments) and to favor others which minimize the number of different segments: smaller the number of different segments, smaller the time for finding the placement of the corresponding processing elements.

Figure 7.a is an illustrating example of the result of the FRAP placement. The full FPGA area has been partitioned into three convenient areas. In the convenient area 1, an horizontal snake is made of two different segments, segment of shape A and segment of shape B. The

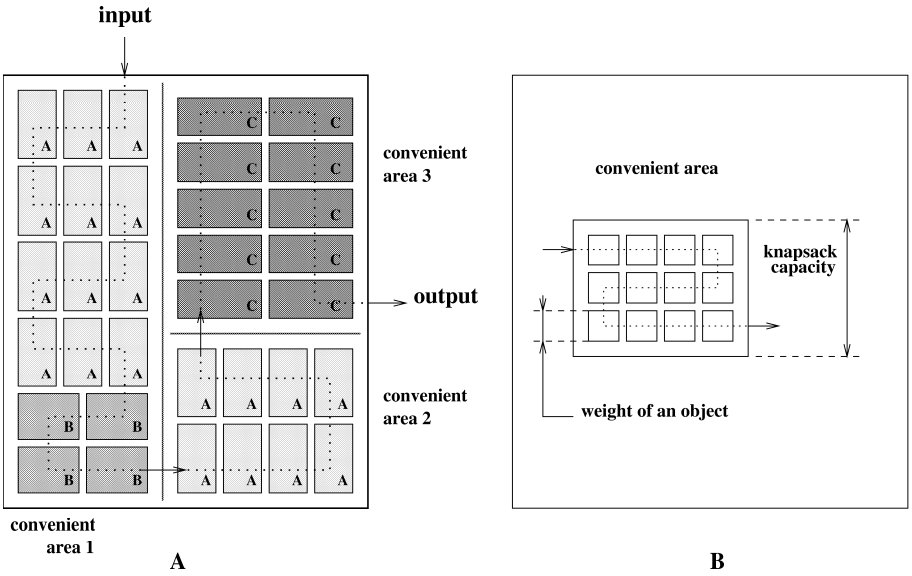


Figure 7: A: illustrating placement produced by FRAP. B: Knapsack problem correspondence

convenient area 2 is also an horizontal snake made only with a processing element of shape A. The convenient area 3 is a vertical snake made of processing elements of shape C. The overall placement requires of determining placements for the 3 different shapes of the processing element.

4.3 Implementation

FRAP is implemented in JAVA. It takes as input a RTL description of a linear array without placement and outputs an equivalent description consisting of snake placement.

Although FRAP is still under development, the current version is sufficiently operational to make some test and experiments. The knapsack technique ensures optimal solutions and, in the present case, in a very short time. To give an idea, the placement of a linear array with large processing elements (150 LUTs) on a Xilinx Virtex 1000 takes less than two seconds on a 295 MHz Sun UltraSparc workstation.

5 Conclusion and Future Works

We have presented a strategy for placing linear regular arrays onto FPGA components. This strategy uses the knapsack technique and provides fast placement compared to the vendor tools. The speed-up comes mainly from the regular nature of the architecture we focus on, that is, linear arrays of identical processors on which a two-level placement is achieved: (1) a

cell-level placement and (2) an array-level placement. A CAD tool, called FRAP, is currently under development for implementing this strategy.

Even if we can drastically shorten the placement step, the overall place-and-route process remains too long to be included into a compiling framework. It may takes a few tens of minutes up to a few hours to achieved a suitable routing, that is definitely too long for programmers who are used to a faster compiling process. Consequently, the next step is to shorten the routing phase.

As for placement, this step can benefit from regular architecture by duplicating routing pattern of the processor cells. Unfortunately, unlike for placement, this strategy cannot be implemented through a few "routing" directives. It requires a detailed knowledge of the routing resources of the target FPGA as well as direct access to the programming of the routing switches.

The solution we are considering to overcome these problems is based on the concept of "Virtual FPGA". Such a device can be represented as an intermediate level between the actual physical FPGA components and an abstract FPGA representation. This is comparable to the concept of an abstract machine, such as a JAVA machine for example. In the present case, the architecture of the Virtual FPGA would be targeted to efficiently support regular arrays. The advantage of this approach is that programming such a device is independent of the FPGA technology and the structure of the component is perfectly known. On the other hand, it is a much more resource consuming approach compared to programming directly a FPGA component with vendor tools. But as technology scales down, leading to steadily increasing numbers of reconfigurable gates, it may not be a critical limitation. And, perhaps, it may be the price to pay for speeding up and simplifying the programming of reconfigurable systems.

References

[1] P. Quinton and V. V. Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing,* 1(2), 1989.

[2] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems,* 4(1), 1996.

[3] E. Fabiani, D. Lavenier and L. Perraudeau. Loop Parallelization on a Reconfigurable Coprocessor. In *WDTA'98 : Workshop on Design, Test and Applications,* Dubrovnik, Croatia, 1998

[4] C. Mauras. Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques. *PhD thesis, université de Rennes 1,* 1989.

[5] P. Le Moenner, L. Perraudeau, P. Quinton, S. Rajopadhye, T. Risset Generating Regular Arithmetic Circuits with ALPHARD. *MPPS'96: Massively Parallel Computing Systems,* Ischia, Italy, 1996.

[6] R.F. Lyon, Two's complement pipeline multipliers. *IEEE Transaction on Communications*, April 1976.

[7] G.M. Megson and I.M. Bland. Generic systolic array for genetic algorithms. *IEEE Proceeding - Computers and Digital Techniques,* vol. 144(2): 107-121, 1997.

[8] Xilinx. *The Programmable Logic Data Book,* 1997.

[9] I. Vassányi. Implementing processor arrays on fpgas. In *8th International Workshop on Field Programmable Logic and Applications,* Talin, Estonia, 1998

[10] A. Koch. Regular datapath on Field-Programmable Gate-Arrays. *PhD thesis, Technical University Braunschweig,* 1997

[11] T. J. Callahan, P. Chong, A. DeHon and J. Wawrzynck. Fast module mapping and placement for datapath in fpgas. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays* ACM, 1998

[12] J. M. Emmert, A. Randhar and D. Bhatia. Fast Floorplanning for fpgas. In *8th International Workshop on Field Programmable Logic and Applications,* Talin, Estonia, 1998

[13] S. Martello and P. Toth. Knapsack Problems: Algorithms and Computer Implementation. *John Wiley and Sons,* 1990.

[14] R. Bellman. Dynamic Programming. *Princeton University Press,* Princeton, NJ, 1957.