An Evolutionary Approach to Multi-FPGAs System Synthesis

F. Fernández de Veja[1], J.I. Hidalgo[2], J.M. Sánchez[1], and J. Lanchares[3]

[1] Departamento de Informática, Centro Universitario de Mérida, Universidad de Extremadura, C/ Sta Teresa de Jornet, 38 - 06800 Mérida, Spain
fcofdez@unex.es, http://atc.unex.es/pacof
[2] Departmento de Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid, C/ Juan del Rosal, 8 -28040 Madrid Spain
hidalgo@dacya.ucm.es, http://www.dacya.ucm.es/hidalgo
[3] Departamento de Informática, Escuela Politécnica, Universidad de Extremadura, Spain
[4] Departmento de Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid, C/ Juan del Rosal, 8, Spain,
julandan@dacya.ucm.es

In this chapter we explain in detail a methodology for Multi-FPGA systems (MFS) design. MFSs are hardware platforms used for a great variety of applications, including dynamically re-configurable hardware applications, digital circuit emulation, and numerical computation. There are a lot of MFS not only academical, but also commercial implementations. We describe a set of techniques based on evolutionary algorithms (EA), and we show that they are capable of solving all of the design tasks (partitioning, placement and routing). Firstly a hybrid compact genetic algorithm (HcGA) solves the partitioning problem and then genetic programming (GP) is used to obtain a solution for the two remaining tasks.

7.1 Introduction

Field Programmable Gate Arrays (FPGAs) are integrated devices used on the implementation of digital circuits by means of a configuration or programming process. There are different manufacturers and several kind of FPGAs are available. We will focus on those called island-based FPGAs. This model includes three main components: configurable logic blocks, input-output blocks and connection blocks (see figure 7.1). Configurable logic blocks (CLBs) are used to implement all the logic circuitry. They are positioned in a matrix way in the device, and they have different configuration possibilities. Input-output blocks (IOBs) are responsible for connecting the circuit implemented by the CLBs with any external system. The third class of components are connection blocks (switch-boxes and interconnection lines). They are the elements available for the designer to make the internal routing of the circuit. In most occasions we need to use some of the CLBs to accomplish the routing [1].

When the size of an FPGA is not enough to implement large circuits, the designer must think on higher reconfigurable platforms, in other words, on the use of Multi-FPGA system (MFS) [2]. These systems can eventually include, in addition to several FPGA devices, memories and other hardware elements. MFS are used for dynamically re-configurable hardware applications [3][4], digital circuit emulation [5], numerical computation [6], etc [7] [8]. The two most widely used topologies are the mesh and crossbar types. Mesh MFSs have simple routing methodologies, an easy expandability, FPGAs are connected in the nearest-neighbor pattern, and all devices are used for the same functionality. Fig. 7.2 (a) represents a mesh-topology MFS. A Crossbar MFS model is depicted on figure 7.2 (b). On this style, FPGAs are separated into logic and routing chips. Crossbar distributions are normally designed for some specific problems, but they usually waste logic and routing resources. For these reasons we have focused on mesh topologies.

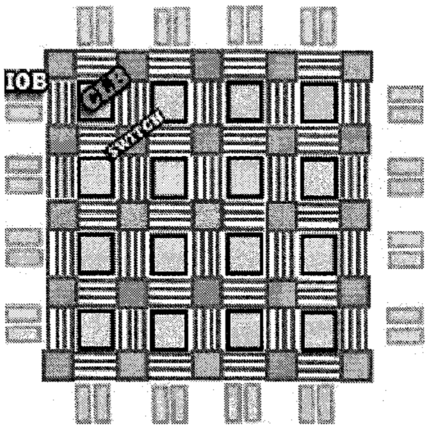
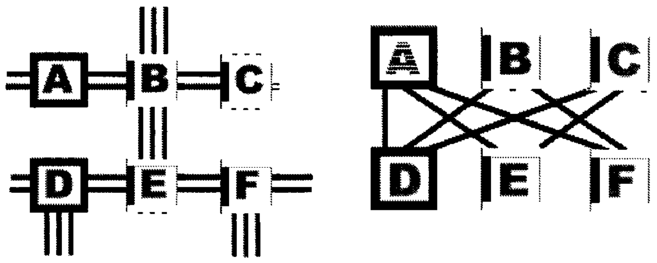


Fig. 7.1. General structure of an island-based FPGA

MFSs design flow has three major tasks: partitioning, placement and routing (see figure 7.3). Frequently two of these tasks are tackled together, because when accomplishing the partitioning, the placement must be considered or vice versa in order to obtain the optimal implementation. In this chapter a methodology, based on evolutionary computation, for the automation of the



(a) crossbar (b) topologies

Fig. 7.2. Multi-FPGA Mesh

whole design flow is explained. There are two separated steps: First, the partitions of the circuit are obtained. During the first stage of the design flow, we also assign a partition (portion of the circuit) to each FPGA. The second step is devoted to place and route the circuit using the FPGA resources. Two different evolutionary algorithms are used: a hybrid compact genetic algorithm (HcGA) for the partitioning step and the genetic programming (GP) technique for the routing and placement step. The experimental results have been obtained in the basis of a real board made up of 8 FPGA (see later 7.11).

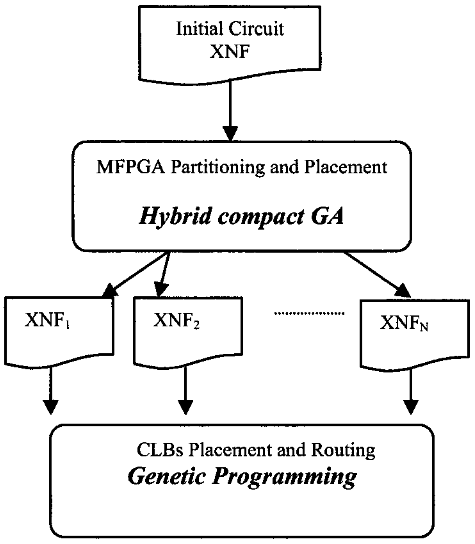


Fig. 7.3. MFS Design Flow

The rest of the chapter is organized as follows: section 7.2 shows an overview about Evolutionary Algorithms, the Compact Genetic Algorithm and Genetic Programming. Section 7.3 describes the partitioning methodology, while section 7.4 shows how the design process within the FPGAs - including the placement and routing steps- has been performed. Section 7.5 contains the experimental results and finally we offer our conclusions in section 7.6.

7.2 Evolutionary Algorithms

Several decades ago, some researchers begun to explore how some ideas taken from nature could be adapted and harnessed for solving well-know difficult problems. Among the concepts borrowed from nature, natural evolution demonstrated from the beginning how simple but also brittle ideas can be helpful for devising new ways of solving difficult problems. Among the techniques that arose under the umbrella of natural evolution, Genetic Algorithms (GAs) [9], Evolutionary Programming [10] and Evolution Strategies [11, 12] have pioneered, matured and demonstrated its usefulness. More recently, John Koza [13] presented Genetic Programming (GP) a new technique that aims at automatically developing computer programs. Koza employed Lisp expressions for evolving programs, and this has favored the use of tree-like data structures in GP, although some researchers have sometimes employed different alternatives. Basically, any EA -including GP- can be described by means of algorithm 7.1.

Algorithm 7.1 Evolutionary algorithm

```
1. Initialize the population.
2. Evaluate all of the individuals in the population and assign a fitness value to
   each one.
3. Select individuals in the population using the selection algorithm.
4. Apply genetic operations to the selected individuals.
5. Insert the result of the genetic operations into the new population.
6. If the population is not fully populated go to step 3.
7. If the termination criterion is reached, then present the best individual as the
   output. Otherwise, replace the existing population with the new population and
   go to step 3.
```

We notice from the algorithm that an evaluation process is performed in step 2. Therfore, for evaluating individuals, a fitness function has to be implemented. This function is in charge of computing a fitness value for the individual under evaluation. The fitness value is proportional to the quality of the individual. The selection operation usually takes into account the fitness value of individuals, and select with higher probabilities those with larger fitness values. Finally, we must point out that crossover and mutation are the genetic operations applied to the individuals selected. Crossover operator takes a couple of individuals, that act like parents, and exchange some of their information, thus creating a couple of new descendant individuals, that share information from both parents. On the other hand, the mutation operation, randomly mutate some of the information contained in the individual to which the operation is applied. Depending on the kind of EA employed, different data structures for encoding candidate solutions -individuals- might be employed. Typically, individuals are encoded by means of bit or integer strings when using GAs, while tree structures are employed for GP.

7.2.1 The Compact Genetic Algorithms

In [14] a compact Genetic Algorithm (cGA) has been proposed. It does not manage a population of solutions but only mimics its existence and it simulates the order-one behavior of a simple GA with uniform crossover. The cGAs' authors do not propose it as an alternative algorithm but it can be used to quickly estimate the "difficulty" of a problem. A problem is easy if it can be solved with a cGA exploiting a low selection rate. The more the selection rate must be increased to solve the problem, the more it has to be considered difficult.

The idea on which the cGA is based was primarily inspired by the *random walk* model, proposed to estimate GA convergence on a class of problems in which there is no interaction among the building blocks constituting the solution [15]. Other concepts that inspired the cGA were *Bit-based Simulated Crossover* (BC) [16] and *Population-Based Incremental Learning* (PBIL) [17]. The cGA represents the population by means of a vector of values $p_i \in [0,1], \forall i = 1,\ldots,l$, where l is the number of alleles needed to represent the solutions. Each value p_i measures the proportion of individuals in the simulated population which have a zero (one) in the i^{th} locus of their representation. By treating these values as probabilities, new individuals can be generated and, based on their fitness, the probability vector updated accordingly in order to favour the generation of better individuals.

The initial probabilities values, p_i, are set to 0.5 to represent a randomly generated population in which the value for each allele has equal probability. At each iteration, the CGA generates two individuals on the basis of the current probability vector and compares their fitness. Lets W be the representation of the individual with better fitness, and L the one of the individual whose fitness was worse. The competitor representations are used to update the probability vector at step $k+1$ in the following way:

$$p_i^{k+1} = \begin{cases} p_i^k + 1/n & \text{if } W_i = 1 \wedge L_i = 0 \\ p_i^k - 1/n & \text{if } W_i = 0 \wedge L_i = 1 \\ p_i^k & \text{if } W_i = L_i \end{cases} \tag{7.1}$$

where n is the dimension of the population simulated, and W_i (L_i) is the value of the i^{th} allele of W (L). The cGA ends when the values of the probability vector are all equal to 0 or 1. At this point the vector p itself represents the final solution. Note that the cGA evaluates an individual by considering its whole chromosome. At each iteration, some alleles of solution W might not belong to the optimal solution of the problem, and the correspondent probability values wrongly modified. For example, consider the OneMax problem, in which the related fitness function computes the number of bits set to 1 of a binary string. Lets $a = 10110$ and $b = 01010$ be the two competitors. String a clearly is the individual with better fitness. The first and third element of the probability vector are thus increased by $1/n$, the fourth and fifth elements remain unchanged, while the second element is incorrectly decreased by $1/n$.

Algorithm 7.2 Pseudo-code of the CGA for the TSP.

```
Program TSP_CGA
  begingroup
     Initialize(P,method);
     F_best := INT_MAX;
     count := 0;
     repeat
        S[1] := Generate(P);
        F[1] := Tour_Lenght(S[1]);
        idx_best := 1;
        for k := 2 to s do
           S[k] := Generate(P);
           F[k] := Tour_Lenght(S[k]);
           if (F[k] < F[idx_best]) then idx_best := k;
        end for
        for k := 1 to s do
           if (F[idx_best] < F[k]) then Update(P,S[idx_best],S[i]);
        end for
        if (F[idx_best] < F_best) then
           count := 0;
           F_best := F[idx_best];
           S_best := S[idx_best];
        else
           Update(P,S_best,S[idx_best]);
           count := count + 1;
        end if
     until (Convergence(P) OR count > CONV_LIMIT)
     Output(S_best,F_best);
  end
```

In order to represent a given population of n individuals, the cGA updates the probability vector by a constant value equal to $1/n$. Only $\log_2 n$ bits are thus needed to store the finite set of values for each p_i. The CGA therefore requires $\log_2 n * l$ bits with respect to the $n * l$ bits needed by a classic GA. Larger population dimension can be exploited without significantly increasing memory requirements, but only slowing CGA convergence. This peculiarity makes the use of CGAs very attractive to solve problems for which the huge memory requirements of GAs is a constraint.

To solve problems higher than order-one GAs with both higher selection rates and larger population sizes have to be exploited [18]. The cGA selection pressure can be increased by modifying the algorithm in the following way: **(1)** generate at each iteration s individuals from the probability vector instead of two; **(2)** choose among the s individuals the one with best fitness and select as W its representation; **(3)** compare W with the other $s-1$ representations and update the probability vector accordingly. The other parts of the algorithm remain unchanged. Such an increase on the selection pressure helps the cGA to converge to better solutions since it increases the survival probability of higher order building blocks [14]. Algorithm 7.2 shows a pseudocode of the cGA for the TSP problem.

7.2.2 Genetic Programming

One of the difference between GP and other EAs is that fitness values are to be computed by evaluating computer programs. If we consider that individuals -programs- are encoded by means of tree like structures (see figure 7.4), each program is made up of internal nodes -functions- and terminals -the leaves of the tree. Which functions and terminals are of interest for the problem that is to be solved is decided by the researcher, and usually varies largely from a problem to another. For instance, if we employ GP for solving a symbolic regression problem, we may choose arithmetic functions for the function set, while if we apply GP for programming a robot, some primitives that allows to move the robot along several directions could make up the function set. The terminal set are usually made up of the constant values and parameters employed by the functions included in the terminal set. Therefore, the first concern for GP practitioners is to appropriately define the function and terminal sets. This means that even when the solution for the problem to be addressed is not known, one must be sure that the solution can be found using the functions and terminals selected.

Genetic operators applied in GP are similar to those employed with any other Evolutionary Algorithm. One of the main differences is due to the kind of data structures employed. When crossover is applied to a couple of individuals, two new descendants are obtained by exchanging some randomly chosen subtrees from each of the parents (see figure 7.5). On the other hand, mutation operator generates a new individual by substituting a randomly chosen subtree from the parent, by a new one that is also randomly generated (see figure 7.6). Although other possibilities are available, the previously described ones are the simplest and most widely employed versions of the genetic operators.

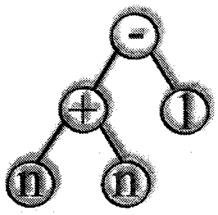


Fig. 7.4. Individuals are encoded by means of trees in Genetic Programming.

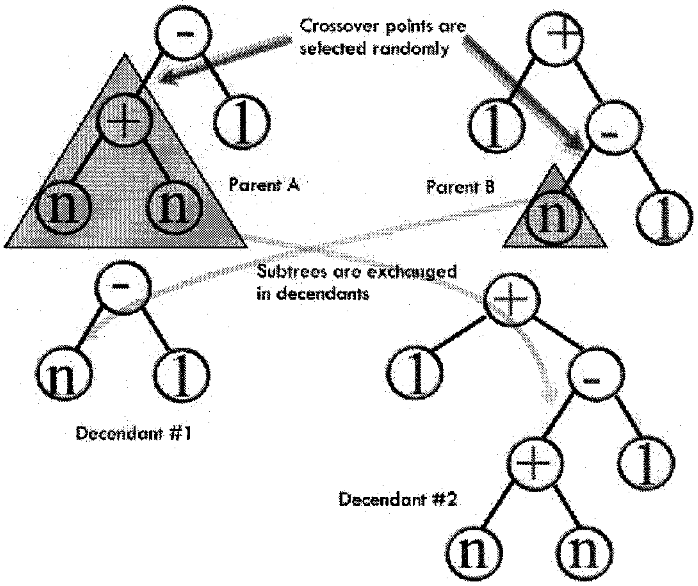

Crossover points are selected randomly
Parent A
Parent B
Subtrees are exchanged in decendants
Decendant #1
Decendant #2

Fig. 7.5. Crossover operation.

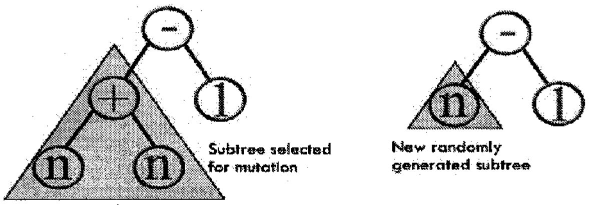


Fig. 7.6. Mutation operation.

Once all of the above components are integrated within the GP algorithm -that is basically the same described in algorithm 7.1-, it can be applied to any optimization problem. In section 7.4 we show how GP has been applied for solving the problem of Placement and routing circuits on FPGAs. A wider description of Genetic Programming can be found in [19].

7.3 MFS partitioning and FPGA assignment

In this section we present the first stage of the design flow. We describe different techniques and algorithms presented in several papers. Most of the previous approximations do not preserve the structure of the circuit or use a difficult encoding. For example Laszewski and M1/4hlenbein implemented a parallel GA which solves the graph partitioning problem with an easy encoding, but the solutions do not preserve the structure of the circuit, and that is a key issue if we want to minimize the delays of the partitioned circuitn [20]. Alpert uses a GA for improving another partitioning algorithm with good results for bi-partitions [21] . An exception, concerning the structure, is the approximation made by Hulin [22]. The approximation used here solves these problems. It is adaptable and can be modified for using in other graph partitioning problems with few changes, it is parallelizable (the method is intrinsically parallel, because it uses a genetic algorithm as a tool for optimization), and in addition, the evaluation of the fitness function can be parallelized very easily. The algorithm also preserves the structure of the circuit and it detects those parts of the graph which are independent.

7.3.1 Methodology

partitioning deals with the problem of dividing a given circuit into several parts, called partitions, in order to be implemented on a MFS. The partitions are obtained and each partition is assigned to a different FPGA within the board. We use a 8-FPGA Mesh topology board, so we must bear in mind several constraints related to the board. Some, and usually most important, of these constraints are the number of available I/O pins on each FPGA and logic capacity. FPGA devices have a much reduced number of pins when compared with their logic capacity. In addition we must connect parts of the circuit that are placed on non-adjacent FPGAs, and for this task we have to use some of the available pins. Partitioning appears in a lot of design automation design problems, and most of the research related to MFS partitioning were adapted from other VLSI areas [23]. For this specific board we have developed a new methodology. We apply the graph theory to describe a given circuit, and then a compact genetic algorithm (cGA) with a local search improvement is applied with a problem-specific encoding. This algorithm not only preserves the original structure of the circuit but also evaluates the I/O-pins consumption due to direct and indirect connections between FPGAs. The MFS placement or FPGA assignment is done by means of a fuzzy technique. We have used the partitioning93 benchmarks [24], described in the Xilinx Netlist Format (XNF), a netlist description language [25].

7.3.2 Circuit Description

Some authors use hyper-graphs as the way of representing a circuit, but there are also some approximations, which use graphs [26]. We have thus, employed an undirected graph representation to describe the circuit. This representation permits an efficient encoding of the compact genetic algorithm and a direct encoding of the solutions using this code.

Hidalgo et al. [27] describe a method that uses the edges of a graph to represent k-way partitioning solutions. They transform the netlist circuit description into a graph, and then operate with its spanning tree. A spanning tree of a graph is a tree, which has been obtained selecting edges from this graph. One of the properties of a spanning tree is that if n edges are suppressed, $n-1$ isolated trees are obtained. As we are treating a k-way partitioning problem, $k-1$ edges of the spanning tree are selected and eliminated in order to obtain k partitions of the original circuit. The partitions are represented by the deleted edges and a hybrid compact genetic algorithm (HcGA) works under this representation to obtain the best partitioning accordingly to the board constraints previously explained. Based on the previous statement, a specific algorithm to address the partitioning and placement problems in MFS systems can be used. The algorithm, which is also adaptable to different boards and devices, preserves the main structure of the circuit and, by means of a fuzzy technique, evaluates the IO pins consumption due to not only direct, but also indirect connections between FPGAs within the MFS (an 8-FPGA board).

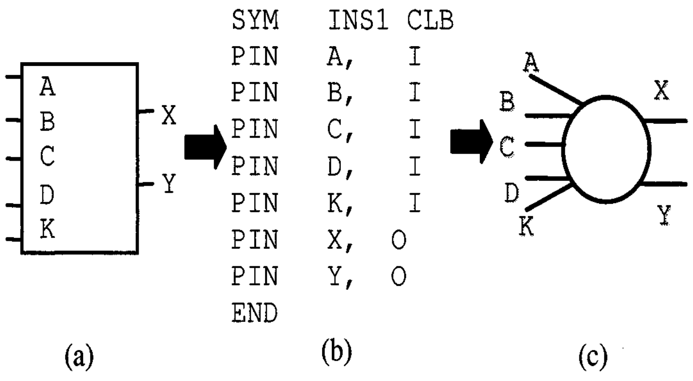

```
SYM  INS1 CLB
PIN  A,   I
PIN  B,   I
PIN  C,   I
PIN  D,   I
PIN  K,   I
PIN  X,  O
PIN  Y,  O
END
```

Fig. 7.7. An example of a CLB described in (a)block, (b)XNF,and (c)graph formats.

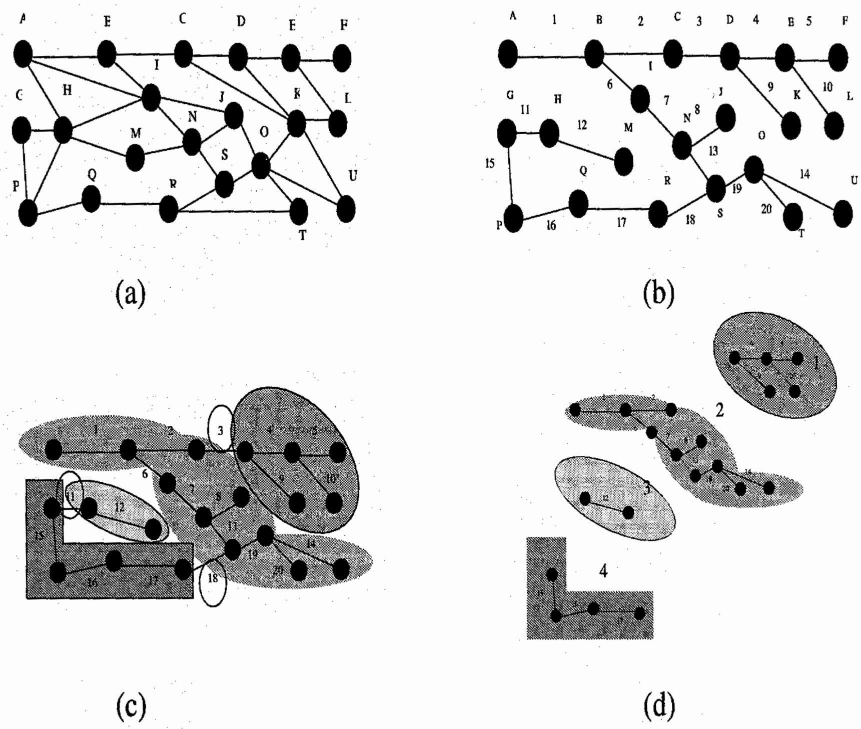


Fig. 7.8. An example of the partitioning process for 4 FPGAs.

The main objective is to solve the circuit partitioning problem and to obtain a set of portions or partitions of the original circuit suitable for the implementation over a single FPGA. The partitioning process is targeted to a device board which has their devices connected in a 4-way mesh topology [2]. So, the method works as follows. First a graph representing the circuit netlist description is obtained. Fig. 7.7 shows the equivalence between an XNF netlist description of a Configurable Logic Block and a graph. After that a spanning tree of that graph is randomly selected, from this tree we select $k-1$ edges and we eliminate them in order to obtain a $k-way$ partition. The partitions are represented by the deleted edges. In Fig. 7.8 we can see an example of the partitioning process. Starting from the circuit graph (a), we get its spanning tree (b) using the Kruskal algorithm [26]. From it, we select the necessary edges and finally we obtain the partitions (c). The figure represents an example for four FPGA devices, so we select only 3 edges of the tree. Once the partitions have been obtained the graph representation can be transformed into a XNF file for each partition and then these files, with the necessary additional information, can be implemented on each FPGA (see Fig. 7.9).

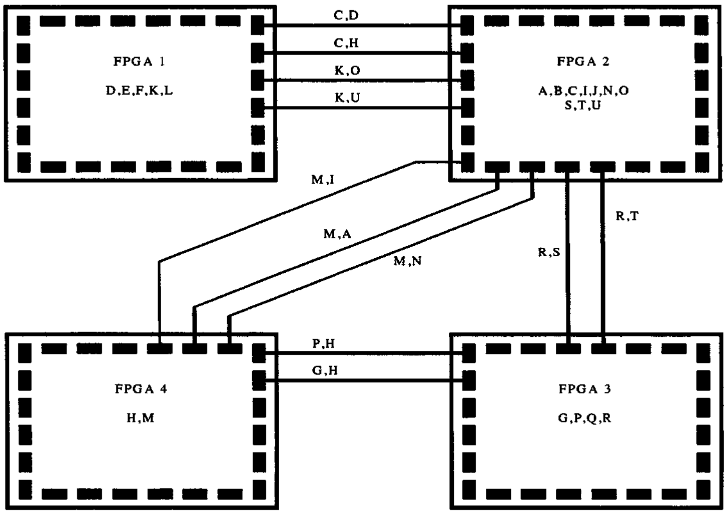


Fig. 7.9. An example of a post-partitioning implementation using 4 FPGAs.

It is important to note that when accomplishing the transformation we should work with the whole graph instead with its spanning tree. This is because the information related to connections is included in the graph and the spanning tree only works with some of them. It is necessary to determine the optimum distribution of the CLBs on the different available FPGAs. An optimum distribution has a minimal cost and guarantees the internal routability of each FPGA.

7.3.3 Genetic Representation

The evaluation process tell us the goodness of the solutions by means of a fitness function. The main task of the HcGA is to solve the partitioning while attending some board requirements related to IO pins and logic blocks (called CLBs on Xilinx's devices). The fitness function guides the search of the algorithm, so it must minimize the number of cutting edges (that is the connections between FPGAS of the MFS) and in addition, it must distribute the blocks uniformly among the FPGAs. So we have a multi-objective genetic algorithm problem. This problem is well known and a number of non-genetic and genetic algorithms have been implemented for its resolution [28] [29]. One of the techniques commonly used is the use of added functions which include weighted sum methods, where the user assigns a weight to each objective and the total fitness is the sum of all weighted fitness values. Nowadays a lot of multi-objective techniques are available for the designer to adapt those partitioning problems.

In order to design a cGA for Multi-FPGA Partitioning we adopted the *edge* representation previously commented and we consider the frequencies of the edges occurring in the simulated population. A vector V of dimension equal to the number of nodes minus one was used to store these frequencies. Each element v_i of V represents the proportion of individuals whose partition use the edge e_i. The vector elements v_i were initialized to 0.5 to represent a randomly generated population in which each edge has equal probability to belong to a solution. In Algorithm 7.3 the pseudo code of a cGA to solve Multi-FPGA partitioning is shown.

Algorithm 7.3 Pseudo-code of the cGA for Multi-FPGA Partitioning.

```
Program Multi-FPGA-cGA
  begin
    Initialize(V);
    F_best := INT_MAX;
    count := 0;
    repeat
       S[1] := Generate(V);
       F[1] := Partition(S[1]);
       idx_best := 1;
       for k := 2 to s do
          S[k] := Generate(V);
          F[k] := Partition(S[k]);
          if (F[k] < F[idx_best]) then idx_best := k;
       end for
       for k := 1 to s do
          if (F[idx_best] < F[k]) then Update(V,S[idx_best],S[i]);
       end for
       if (F[idx_best] < F_best) then
          count := 0;
          F_best := F[idx_best];
          S_best := S[idx_best];
       else
          Update(V,S_best,S[idx_best]);
          count := count + 1;
       end if
    until (Convergence(V) OR count > CONV_LIMIT)
    Output(S_best,F_best);
  end
```

After the initialization phase an individual is generated and its fitness value is computed. Then, according to the selection pressure adopted $s-1$ individuals are generated, evaluated and the best individual is carried out. The last is used to update the probability vector V according to Equation 7.1. Moreover, the best individual generated in the current iteration (S[idx_best]) is compared with the best individual found until now (S_best) and V is updated accordingly. The cGA proposed in [14] ends when the values of the probability vector are all equal to 0 or 1. Since in our tests such a condition was rarely achieved we introduced a supplementary end condition which limits the maximum number of generations occurring without an improvement of the best solution achieved (see algorithm 7.3). Reached such a limit the execution is terminated and the best individual found is returned as final solution.

The cGA (and also the HcGA) uses the encoding presented in section 7.3.2 which directly represents solutions to the partitioning problem. As we have said, the code is based on the edges of a spanning tree. We have seen above how the partition is obtained by the elimination of some edges. A number is assigned to every edge of the tree. Consequently, for a k-way partitioning problem a chromosome will have k-1 genes, and the value of these genes can be any of the order values of the edges. For example, chromosome (3 14 26 32 56 74 89) for a 8-way partitioning, represents a solution obtained after the suppression of edge numbers 3, 14, 26, 32, 56, 74, and 89 from a spanning tree. So the alphabet of the algorithm is: $\Omega = \{0, 1 \ldots, n-1\}$ where n is the number of vertexes of the target graph (circuit), because the spanning tree has $n-1$ edges.

7.3.4 Hybrid Compact Genetic Algorithm

A Hybrid cGA (HcGA) uses non-evolutionary algorithms for local search, that is, to improve good solutions found by the cGA. When designing a cGA for MFS partitioning, a vector (V), with the same dimension as the number of nodes minus one, stores the frequencies of the edges occurring in the simulated population. Each element vi of V represents the proportion of individuals whose partition use the edge ei. Following the original cGA, the vector elements vi were initialised to 0.5 to represent a randomly generated population in which each edge has equal probability to belong to a solution [14]. Sometimes it is necessary to increase the selection pressure rate Ps, (the number of individuals generated on each iteration) to reach to good results with a Compact Genetic Algorithm. A value for Ps near to 4 has shown to be a good value for MFS partitioning. It is not to be recommended a large increasing of this value, because the computation time will grow drastically. Additionally, for some problems we need a complement to cGA in order to solve them properly. We can combine heuristics techniques with local search algorithms to obtain this additional tool called hybrid algorithms. We have implemented a cGA with local search.

In [30] a compact genetic algorithm for MFSs partitioning was presented, and in [31] a Hybrid cGA was explained. Authors combine a cGA with the Lin-Kernighan (LK) local search algorithm, to solve Traveling Salesman Problems (see Algorithm 7.2). The cGA part explores the most interesting areas of the search space and LK task is the fine-tuning of those solutions obtained by cGA. Following this structure, but changing the local search method, we can implement a hybrid cGA for MFS partitioning. Ideally, a local search algorithm must try to perform the search process as exhaustively as possible. Unfortunately, in our problem this also implies an unacceptable amount of computation. Therefore, we have employed a local search heuristic each certain number (n) of iterations and we need to study the value of n to keep the algorithm search in good working order. After empirically studying the local search frequency, we have obtained that n must be assigned a value between 20 and 60, with an optimal value (that depends on the circuit benchmark) near to 50. So for our experiments we fixed the local search frequency n to 50 iterations, i.e. we develop a local search process every 50 iterations of the cGA.

Now it is necessary to define a new concept, neighbouring. We have mentioned that a chromosome has $k-1$ genes for a k-way partitioning, and the value of these genes are the edges that are removed from the spanning tree representing the circuit when looking for a solution.

Definition.

solution A is a neighbour solution of B (and B is a neighbour solution of A) if the difference between their chromosomes is just one gene.

Our local search heuristic explores only one neighbour solution for each gene, that is $k-1$ neighbouring solutions of the best solution every n iterations. The local search process works as Algortihm 7.4 explain [32].

Although only a very small part of the solution neighbourhood space is explored, the performance of the algorithm improves significantly (in terms of quality of solutions) without degrading drastically its total computation time. In order to clarify the explanation about the proposed local search method we can see an example. Let us suppose a graph with 12 nodes and its spanning tree, for a 5-way partitioning problem (i.e. we want to divide the circuit into five parts). As we have explained, we will use individuals with 4 genes. Let us also suppose a local search frequency (n) of 50 and that after 50 iterations we have reached to a best solution represented by:

$$BS = (3, 4, 6, 7) \tag{7.2}$$

The circuit graph has 12 nodes, so its spanning tree is formed by 11 edges. The whole set of possible edges to obtain a partitioning solution is called E:

$$E = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \tag{7.3}$$

In order to generate TS1 we need to know the available edges ALS for random selection, as we have said, we eliminate the edges within BS from E to obtain ALS:

$$ALS = \{0, 1, 2, 5, 8, 9, 10\} \tag{7.4}$$

Algorithm 7.4 Local search algorithm for MFS partitioning HcGA.

```
1. Every n iterations, we obtain the best solution up to that time (BS).To obtain BS:
   a) first we explore the compact GA probability vector and select the k-1 most used genes (edges) to form MBS (vector best individual).
   b) The best individual generated up to now (GBS) (similar to elitism) is also stored.
   c) The best individual between MBS and GBS (i.e. which of them has the best fitness value) will be BS.
2. the first random neighbour solution (TS1) to BS is generated substituting the first gene (edge) of the chromosome by a random one, not present in BS.
3. Calculate the fitness value of BS (FVBS) and the fitness value of TS1 (FVTS1)
4. Compare If FVTS1 is better than FVBS, if so TS1 is dropped to BS and the initial BS is eliminated, otherwise TS1 is eliminated
5. Repeat the same process using the new BS and with the second gene, to generate TS2
6. If the fitness value of TS2 (FVTS2) is better than the present FVBS then TS2 will be our new BS or, if FVTS2 is worst than FVBS, there will be no change in BS.
7. Repeat last step for the rest of the genes untill the end of the chromosome (that is, k-1 times for a k-way partitioning).
```

Now we randomly select an edge (suppose 0) to build TS1substituting it by the first gene in BS:

$$TS1 = (0, 4, 6, 7) \tag{7.5}$$

The third step is the evaluation of $TS1$ (suppose $FVTS1 = 12$) and comparing (suppose a minimization problem) with $FVBS$ (suppose $FVBS = 25$). As FVTS1 is better than FVBS, TS1 will be our new BS and the original BS is eliminated. Those changes also affect to ALS because our new ALS is:

$$ALS = \{1, 2, 3, 5, 8, 9, 10\} \tag{7.6}$$

Table 7.1 represents the rest of the local search process for this example.

7.4 Placement and Routing on FPGAs

Once the first step has been carried out, we have several partitions. Each partition - that is in charge of a small circuit - have to be implemented independently in a different FPGA. Finally, all the FPGAs will be connected together, thus obtaining the global circuit. Even when much research has been done on the automatic generation of digital and analogue circuits, we will review now some proposals that are related with the idea of applying evolutionary algorithms to the problem we are addressing, and with the way circuits are encoded.

Table 7.1. Local Search example

i	ALS	BS	FV	Random gene	TS	FV	New Bs
1	0,1,2,5,8,9,10	3,4,6,7	25	0	0,4,6,7	12	0,4,6,7
2	1,2,3,5,8,9,10	0,4,6,7	12	1	0,1,6,7	37	0,1,6,7
3	1,2,3,5,8,9,10	0,4,6,7	12	9	0,4,9,7	10	0,4,9,7
4	1,2,3,5,6,8,10	0,4,9,7	10	8	0,4,8,9	11	0,4,9,7
	Pre-Local Search	Best Solution:		3,4,6,7			
	Post-Local Search	Best Solution:		0,4,9,7			

A given circuit, with wires, gates and connections, can be considered as a graph. Several papers have dealt with the problem of encoding graphs, i.e. circuits, when working with GA and GP [33]. Sometimes new techniques have been developed to do so. For instance, Cartesian Genetic Programming [34] is a variation of GP which was developed for representing graphs, and shows some similarities to other graph based forms of genetic programming. Miller et al's aim is to find complete circuits capable of implementing a given boolean function. Nevertheless, we are more interested in physical layout. Our optimisation problem begins with a given circuit description, and the goal is to find out how to place components and wires in FPGAs. Meanwhile we have also developed a new methodology for representing circuits by means of GP with individuals represented as trees.

Other researchers have also applied Evolutionary Algorithm for evolving analogue circuits [33]. Even Koza have employed Genetic Programming for designing and discovering analogue circuits [35], which have eventually been patented. Thompson's research scope is the physical design and implementation of circuits in FPGAs [36]. However, all of them work with analogue circuits, while we are addressing digital ones. Another difference is the kind of evolutionary algorithm employed for solving each problem. Thompson uses GAs while we are using GP (Koza uses GP but not for solving the kind of problem we address here).

There are also other researchers that have addressed problems employing reconfigurable hardware and Genetic Programming. For instance, in [37] authors describe how trees can be implemented and evaluated on FPGAs. But our aim is not to implement a Genetic Programming tool on an FPGA but using GP for physically placing and routing circuits. Therefore, in this second step, we take each of the partitions as the input of the problem, and the goal is to place components and establish connections among them in a different FPGA. Our proposal now is to use Genetic Programming (GP) for solving this task. The main reason behind this choice is the similarity between data structures that GP uses -trees- and the way of describing circuits -graphs. A tree is more convenient than a fix-sized string for describing graphs of any length. In the following sections we describe how graphs are encoded by means of trees.

7.4.1 Circuits encoding using trees

As described in section 7.3, the output for the partitioning algorithm is a set of partitions, and a description of the way they must be connected. Each of the partition includes a circuit that must be implemented in a separate FPGA. Therefore, the main goal for this step is to implement a partition (circuit) into an FPGA. Each of the circuit component has to be implemented into a CLB, and after that previous step, all the CLBs have to be connected according to the circuit's topology. Given that we use tree-based GP in this stage of the methodology, we need a mapping between a graph -circuit- and a tree. Circuits have to be encoded as trees, and any of the trees that GP will generate, should also have an equivalent circuit; the fitness function will later decide if the circuit is correct or not, and its resemblance degree with the correct circuit.

Considering that any of the components of a circuit is simple enough to be implemented employing a CLB from the FPGA, we might describe a circuit employing black boxes, such as is depicted by means of an example in figure 7.10. This means that we only have to connect CLBs from the FPGA according to the interconnection model that a given circuit implements, and then we can configure each of the CLB with the function that each component performs in the circuit. We want to perform this task by using GP. This means that circuits must be described by means of trees -individuals in GP. To do it, we can firstly label each component from the circuit with a number, and then assign components' labels to the ends of wires connected to them (see figure 7.10).

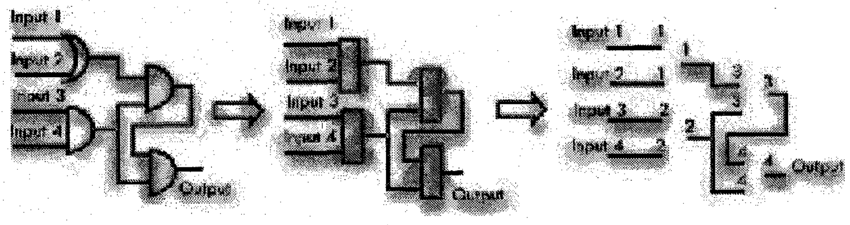


Fig. 7.10. Representing a circuit with black boxes.

We may now describe all the wires by means of a tree by connecting each of the wires as a branch of the tree and keeping them all together in the same tree. By labeling both extremes of branches, we will have all the information required to reconstructing the circuits. Any given tree, randomly generated, will always correspond to a particular graph, regardless of the usefulness of the associated circuit (see figure 7.8). In this proposal, each node from the tree is representing a connection, and each branch is representing a wire. The next stage is to encode the path of wires into an FPGA. Each branch of the tree will encode a wire from the circuit: internal nodes specify switch connections that are traversed by the wire, while the first and last nodes of the branch are employed to connect the wire to an adjacent CLB -by specifying which of the CLB is employed and to which pin is the wire connected.

Each of the branches will include as many internal nodes as required for describing all of the switch connections required for the wire (see figure 7.8). Sometimes, branches will not include any internal nodes. This may happen when an input/output connection is directly attached to any of the CLB from the surrounding area of the FPGA. Only two nodes are required in the branch: the first one specify which IOB is employed, while the second one select the CLB to which it is connected and the wire employed.

Each internal node requires some extra information: if the node corresponds to a CLB we need to know information about the position of the CLB in the FPGA, the number of pin to which one of the ends of the wire is connected, and which of the wires of the wire block we are using; if the node represents a switch connection, we need information about that connection (figures 7.11 and 7.12 graphically depicts how a tree describes a circuit, and the way each branch maps a connection).

It may well happen that when placing a wire into an FPGA, some of the required connections specified in the branch can not be made, because, for instance, a switch block connection has been previously used for routing another wire segment. In this case the circuit is not valid, in the sense that not all the connections can be placed into a physical circuit, and the function in charge of analyzing the tree will apply a high penalty to that individual from the population.

In order for the whole circuit to be represented by means of a tree, we will use a binary tree, whose left most branch will correspond to one of its connections, and the left branch will consist of another subtree constructed recursively in the same way (left-branch is a connection and right-branch a subtree). The last and deepest right branch will be the last circuit connection. Given that all internal nodes are binary ones we can use only a kind of function with two descendants. In the following subsection we describe the GP sets required.

7.4.2 GP sets

When solving a problem by means of GP one of the first things to do once the problem has been analyzed is to build both the function and terminal sets. The function set for our problem contains only one element: F={SW}, Similarly, the terminal set contains only one element T={CLB}. But SW and CLB may be interpreted differently depending on the position of the node

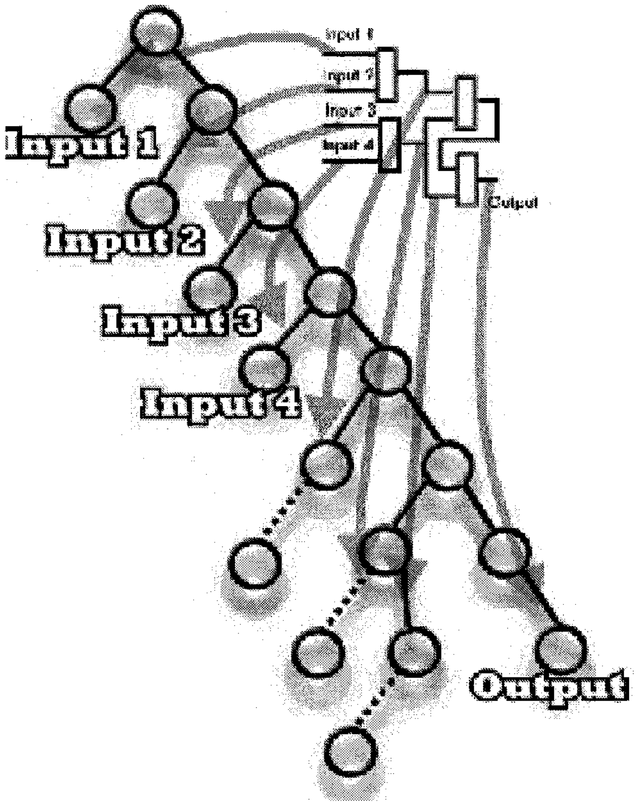


Fig. 7.11. Making connections in the FPGA according to nodes

within a tree. Sometimes a terminal node corresponds to an IOB connection, while sometimes it corresponds to a CLB connection in the FPGA (see figure 7.8. Similarly, an internal node - SW node- sometimes corresponds to a CLB connection (the first node in the branch), while others affects switch connections in the FPGA (internal node in a branch, see figure 7.9). Each of the nodes in the tree will thus contain different information:

- If we are dealing with a terminal node, it will include information about the position of CLBs, the number of pins selected, the number of wires to which it is connected, and the direction we are taking when placing the wire.
- If we are instead in a function node, it will have information about the direction we are taking. This information enables us to establish the switch connection, or in the case of the first node of the branch, the number of the pin where the connection ends.

We can notice in figure 7.8, that wires with IOBs at one of their ends are shorter -only needs a couple of nodes- than those that have CLBs at both

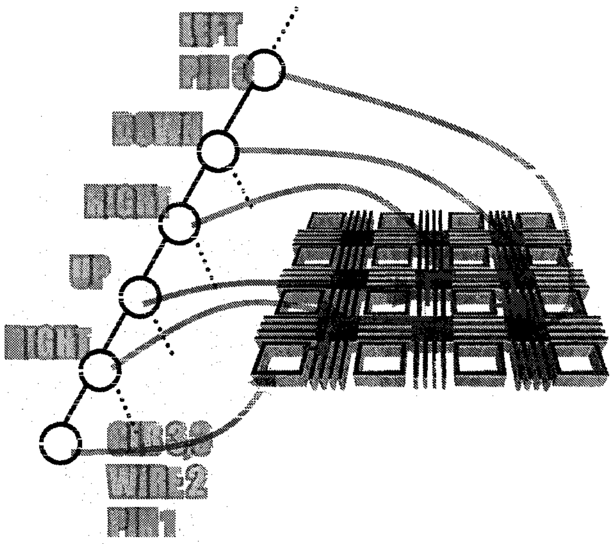


Fig. 7.12. Encoding circuits by means of binary trees. Each branch of the tree describes a connection from the circuit. Dotted lines indicates a number of internal nodes in the branch

ends -they require internal nodes for expressing switch connections-. Wires expressed in the latest position of trees have less space to grow, and so we decided to place IOB wires in that position, thus leaving the first parts of the trees for long wires joining CLBs.

7.4.3 Evaluating Individuals

In order for GP to work, individuals from the population have to be evaluated and reproduced employing the GP algorithm. For evaluating an individual we must convert the genotype (tree structure) to the phenotype (circuit in the FPGA), and then compare it to the circuit provided by the partitioning algorithm. We developed an FPGA simulator for this task. This software allows us to simulate any circuit and checks its resemblance to other circuit. Therefore, this software tool is in charge of taking an individual from the population and evaluating every branch from the tree, in a sequential way, establishing the connections that each branch specifies. Circuits are thus mapped by visiting each of the useful nodes of the trees and making connections on the virtual FPGA, thus obtaining phenotype. Each time a connection is made, the position into the FPGA must be brought up to date, in order to be capable of making new connections when evaluating the remaining nodes. If we evaluate each branch, beginning with the terminal node, thus establishing the first end of the wire, we could continue evaluating nodes of the branch from the bottom to the top. Nevertheless, we must be aware that there are several terminals related to each branch, because each function node has two different descendants. We must decide which of the terminals will be taken as the beginning of the wire, and then drive the evaluation to the top of the branch. We have decided to use the terminal that is reached when going down through the branch using always the left descendant, and evaluate all the nodes traversed from the root of the branch to that terminal (see figure 7.13).

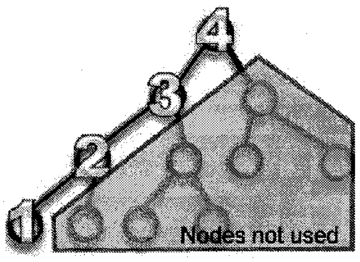


Fig. 7.13. Evaluating a branch of the tree-corresponding to a connection of the circuit. Evaluation order is specified with numbers labelling nodes.

In one sense there is a waste of resources when having so many unused nodes. Nevertheless they represent new possibilities that can show up after a crossover operation (in nature, there always exist recessive genes, which from time to time appear in descendants). These nodes are hidden, in the sense that they do not take part in the construction of the circuit and may appear in new individuals after some generations. If they are useful in solving the problem, they will remain in descendants in the form of nodes that express connections. The fitness function is computed as the difference between the circuit provided and the circuit described by the individual.

7.5 Experimental Results

7.5.1 partitioning and Placement onto the FPGAs

The algorithm has been implemented in C and run on a Pentium 3, 866 MHz with Linux Red Hat 7.3. We have used the MCNC partitioning benchmarks in XNF format. We have supposed that each block of the circuits uses one CLB. We use the Xilinx's 4010 FPGA. 7.2 contains the experimental results. It has five columns which express: the name of the test circuit (Circuit), its number of CLBs (CLB), the number of connections between CLBs (Edges), the number of CLBs used on each FPGA (Distribution) and the CPU time in seconds necessary to obtain a solution for 100 generations of a GA with a population of 501 individuals (T(sec)). There are some unbalanced distributions, because we need to use some resources to pass the nets from one device to another. In addition our fitness function has been developed to achieve two objectives, so that the GA works. To cap it all, the algorithm succeeds in solving the partitioning problem with board constraints.

Fig. 7.14 shows a picture of the board. This card consists of 8 FPGAs of the 4010 family from Xilinx [38] although, these can be replaced by other devices of greater capacity and benefits, just adapting the connections. The FPGAs are connected according to a mesh topology, in other words, they directly connect their next neighbours. The figure shows, in addition to the FPGAs, the electrical power supply and lines for programming them (DIN, DONE, CCLK, INIT, PROGRAM), which allows the configuration by means of an *XChequer* cable from Xilinx. The cable transmits the configuration data to all FPGAs within the board, the transmission frequency is 921 kHz. The speed depends on the used computer, in our case with a PC, a Baud Rate of 115200 can be reached. The power supply used is an ATX computer source. This allows us to have the voltages necessary to feed not only the FPGAS, but also the programming cables such as the *XChequer*. The MFS board also incorporates some jumper pins, for programming and isolation of a group of FPGA within the board. There are also six connectors for expansion of the board using other similar card.

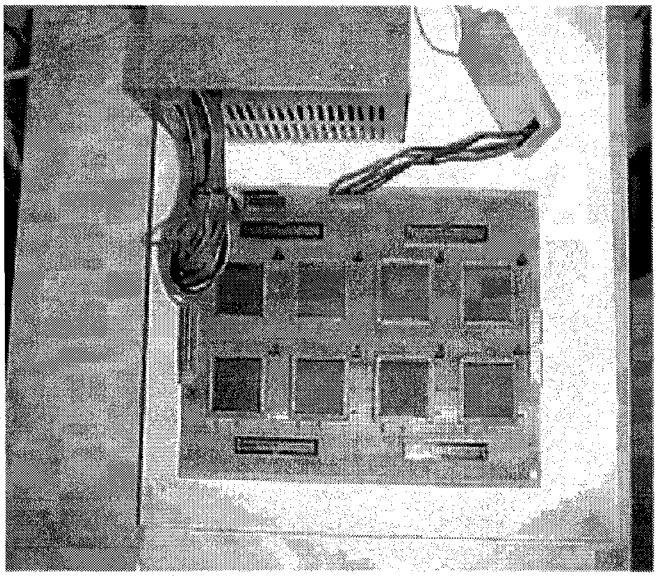


Fig. 7.14. Multi-FPGA board designed for testing the methodology

Table 7.2. Experimental Results for Partitioning and Placement for the 8 -Xilinx 4010 Board

Circuit	CLB	Edges	Distribution	T(sec)
S208	127	200	23,9,16,15,16,18,16,14,	7.83
S298	158	306	24,14,23,27,18,19,21,12	11.91
S400	217	412	15,10,25,34,30,44,30,29	15.71
S444	234	442	17,37,41,15,33,27,29,35	20.63
S510	251	455	28,39,44,23,33,29,30,25	17.64
S832	336	808	32,38,33,38,47,25,65,58	166.94
S820	338	796	41,45,49,49,21,33,41,59	43.95
S953	494	882	45,87,58,63,68,73,67,33	34.83
S838	495	800	30,34,73,25,63,70,34,164	173.71
S1238	574	1127	78,65,70,69,80,98,61,53	483,16
C3540	1778	2115	114,116,119,121,118,128,169,153	1634.00

7.5.2 Inter-FPGA Placement and Routing

Several experiments with different sizes and complexities have been performed for testing the placement and routing process . Fig. 7.15 graphically depicts one of the circuits employed in the series of test of increasing complexity that has been used for validating the methodology (a larger set of experiments and results can be found in [39]). The main parameters employed were the following: Number of generations = 500, Population size: 200, Maximum depth: 30, Steady State Tournament size: 10. Crossover probability=98%, Mutation probability=2%, Creation type: Ramp Half/Half, and elitism.

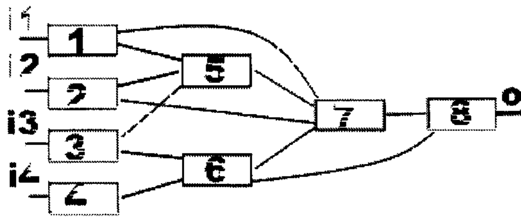


Fig. 7.15. One of the circuits employed for testing the methodology

Fig. 7.16 shows some of the solutions that were obtained with GP- for the circuit described above. A very important fact is that each of the solutions that GP found possesses different features, such as area of the FPGA used, position of the input/output terminals. This means that the methodology could easily be adapted for managing typical constraints in FPGA placement and routing. More solutions found for this and other circuits are described in [39] and [40]. The time required for finding the solution was of some minutes in a 2Ghz Pentium processor. So, the methodology can be successfully employed for routing circuits of larger complexity.

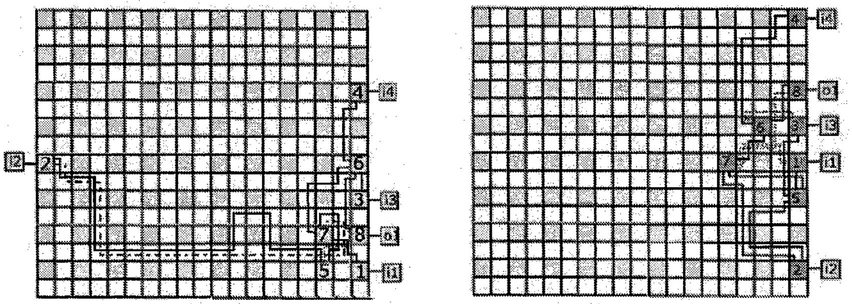


Fig. 7.16. Different solutions obtained by means of GP

7.6 Summary

In this chapter a methodology for circuit design using Multi-FPGA Systems has been presented. We have used evolutionary computation for all the steps of the process. Firstly, an Hybrid compact genetic algorithm was applied on achieving partitioning and placement for inter-FPGA systems and, for the Intra-FPGA tasks Genetic programming was used. This method can be applied for different boards and solves the whole design flow process.

7.7 Acknowledgments

Part of this research has been possible thanks to Ministerio de Ciencia y Tecnologa, research projects number TIC2002-04498-C05-01 and TIC 2002/750.

References

1. S.Trimberger: A reprogrammable gate array and applications. Proceedings of the IEEE **81** (1993) 1030–1040
2. Hauck, S.: Multi-FPGA systems. PhD thesis, University of Washington (1994)
3. Macketanz, R., Karl, W.: Jvx - a rapid prototyping system based on java and fpgas. In Springer-Verlag, ed.: Field Programmable Logic: From FPGAs to Computing Paradigm, Berlin (1998) 99–108
4. Hauck, S.: The roles of fpgas in re-programmable systems. Proceedings of the IEEE **86** (1998) 615–638

5. Baxter, M.: Icarus: A dynamically reconfigurable computer architecture. In: IEEE Symposium on FPGAs for Custom Computing Machines. (1999) 278–279
6. Heywood, M., Zincir-Heywood, A.: Register based genetic programming on fpga computing platforms. In Spinger-Verkag, ed.: Proceedings of EuroGP 2000. (2000) 44–59
7. HArtenstein, R., Kress, R., Reinig, H.: A reconfigurable data-driven alu for xputers. In Press, I., ed.: IEEE Workshop on FPGAs for CUstom Computing Machines. (1994) 139–146
8. Callahan, T., Wawrzynek, J.: Instuction- level parallelism fot reconfigurable computing. In Springer-Verlag, ed.: Field Programmable Logic: From FPGAs to Computing Paradigm, Berlin (1998) 248–257
9. Holland, J.H.: Adpatation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975)
10. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial intelligence through a simulation of evolution. In Maxfield, M., Callahan, A., Fogel, L.J., eds.: Biophysics and Cybernetic Systems: Proc. of the 2nd Cybernetic Sciences Symposium, Washington, D.C., Spartan Books (1965) 131–155
11. Rechenberg, I.: Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. frommann-holzbog, Stuttgart (1973) German.
12. Schwefel, H.P.: Evolutionsstrategie und numerische Optimierung. PhD thesis, Technische Universitat Berlin, Berlin (1975)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
14. G. Harik, F.L., Goldberg, D.: The compact genetic algorithm. Technical Report 97006, University of Illinois at Urbana-Champaign, Urbana, IL (1997)
15. Harik, G.R., Lobo, F.G., Goldberg, D.E.: The compact genetic algorithm. IEEE-EC **3** (1999) 287
16. Syswerda, G.: Simulated crossover in genetic algorithms. In L. D. Whitley, editor,Fondation of Genetic Algorithms 2, pages 38-45, San Mateo, CA, Morgan Kaufmann (1993)
17. Baluja, S.: Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburg, Pennsylvania (1994)
18. Thierens, D., Goldberg, D.: Mixing in genetic algorithms. In: Proceedings of the Fifth International Conference on Genetic Algorithms. (1993) 38–45
19. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag (1998)
20. Laszewski, G., Muhlenbein, H.: A parallel genetic algorithm for the k-way graph partitioning problem. In: 1st inter. Workshop on Parallel Problem Solving from Nature. (1990)
21. C.J. Alpert, L.W.Hagen, A.K.: A hybrid multilevel/genetic approach for circuit partitioning. Technical report, UCLA Computer Science Department (1994)
22. Hulin, M.: Circuit partitioning with genetic algorithms using a coding scheme to preserve the structure of a circuit. In: Lecture Notes in Computer Science, 496, Springer-Verlag (1989) 75–79
23. Alpert, C., Kahng, A.: Recent directions in netlist partitioning: A survey. Technical Report CA 90024-1596, UCLA Computer Science Department (1997)

24. http://vlsicad.cs.ud.edu/: (Cad benchmarking laboratory)
25. Inc, X.: Xnf: Xilinx netlist format. (www.xilinx.xom)
26. Harary, F.: Graph Theory. Addison-Wesley (1968)
27. J.I. Hidalgo, J. Lanchares, R.H.: Graph partitioning methods for multi-fpga systems and reconfigurable hardware based on genetic algorithms. In: Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program. (1999) 357–358
28. Parmee, I.C., Watson, A.H.: Preliminary airframe design using co-evolutionary multi-objective genetic algorithms. In: Proceedings of the 1999 Genetic And Evolutionary Computation Conference, Morgan Kaufmann (1999) 1657–1665
29. C.A.Coello: A comprehensive survey of evolutionary-based multiobjective optimization techniques. Knowledge and Information Systems **1** (1999) 269–308
30. J.I. Hidalgo, R. Baraglia, R.P.J.L.F.T.: A parallel compact genetic algorithm for multi-fpga partitioning. In: PDP20001, 9th Euromicro Workshop on Parallel and Distributed Processing. (2001)
31. R. Baraglia, J.I. Hidalgo, R.P.: A hybrid heuristic for the traveling salesman problem. IEEE Transactions on Evolutionary Computation **5** (2001) 613–622
32. Hidalgo, J.: Multi-FPGA systems partitioning and placement techniques based on Genetic Algorithms. PhD thesis, Universidad Complutense de Madrid (2001)
33. Lohn, J.D., Colombano, S.P.: Automated analog circuit synthesis using a linear representation. Lecture Notes in Computer Science **1478** (1998) 125+
34. Miller, J.F., Job, D., Vassilev, V.K.: Principles in the evolutionary design of digital circuits-part I. Genetic Programming and Evolvable Machines **1** (2000) 7–35
35. Koza, J.R., David Andre, Bennett III, F.H., Keane, M.: Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman (1999)
36. Thompson, A., Layzell, P.J.: Evolution of robustness in an electronics design. In: ICES. (2000) 218–228
37. Seok, H.S., Lee, K.J., Zhang, B.T., Lee, D.W., Sim, K.B.: Genetic programming of process decomposition strategies for evolvable hardware. In: Proceedings of the Second NASA / DoD Workshop on Evolvable Hardware, Palo Alto, California, Jet Propulsion Laboratory, California Institute of Technology, IEEE Computer Society (2000) 25–34
38. Xilinx Corporation (www.xilinx.com)
39. Fernandez de Vega, F.: Distributed Genetic Programming Models with Application to Logic Synthesis on FPGAs. PhD thesis, University of Extremadura (2001)
40. Fernandez, F., Sanchez, J.M., Tomassini, M.: Placing and routing circuits on FPGAs by means of parallel and distributed genetic programming. In Liu, Y., Tanaka, K., Iwata, M., Higuchi, T., Yasunaga, M., eds.: Evolvable Systems: From Biology to Hardware, Proceedings of the 4th International Conference, ICES 2001. Volume 2210 of Lecture Notes in Computer Science., Tokyo, Japan, Springer-Verlag (2001) 204–214