An Evolutionary Approach To Timing Driven FPGA Placement.

R. Venkatraman,
Centre for Electronics Design and Technology,
Indian Institute of Science,
Bangalore, India.

e-mail: rvenkat@ieee.org

Lalit M. Patnaik,
Microprocessor Applications Laboratory,
Indian Institute of Science,
Bangalore, India.

e-mail: lalit@micro.iisc.ernet.in

ABSTRACT:

We propose a novel evolutionary approach to the problem of timing-driven FPGA placement. The method used is evolutionary programming (EP) with incremental position encoded in the population. This uses considerably less memory compared to a method with direct position-encoding for members of the population. The algorithm has been implemented in C++, and the results on MCNC benchmark circuits are presented. The results are superior to those obtained using conventional Simulated Annealing (SA) based approach. The results of an EP-SA approach using the proposed evolutionary programming method are also presented.

1. INTRODUCTION:

FPGAs play an important role in short turn-around time designs. With increase in their utilisation and with advances in technology, timing-driven layout and design using FPGAs have become very important. Conventionally, the issue of timing is addressed at various phases of design - by reducing the number of logic levels during synthesis and technology mapping, and by identifying critical paths and ensuring shorter routing delays for those paths during routing. In FPGA designs, placement plays a crucial role, as it directly influences the timing and routability, primarily because of limited routing resources. Hence the issue of timing has been sought to be incorporated at the placement phase of the CAD flow itself, as it is one of the most important stages in FPGA designs. Timing-driven placement has been studied extensively [5,6,7,9]. Path-based approaches consider timing explicitly, in addition to the physical constraints, while in net-based approaches, the issue of timing is incorporated into a suitable physical constraint.

In this paper, we propose a new Evolutionary Programming (EP)-based solution for the placement of modules on island-style FPGAs. The EP algorithm uses as its members of population, strings denoting incremental encoding of the block positions. The incremental encoding saves considerable memory during the process of evolution. Results obtained with this method are found to be better than those of simulated annealing (SA)-based methods. The rest of the paper is organised as follows. Section-2 deals with problem formulation and evolutionary strategies. Section-3 deals with the proposed algorithm in detail, and section-4 presents the results on technology-mapped MCNC benchmark circuits. We conclude this paper in section-5.

2. PROBLEM FORMULATION AND EVOLUTIONARY STRATEGIES:

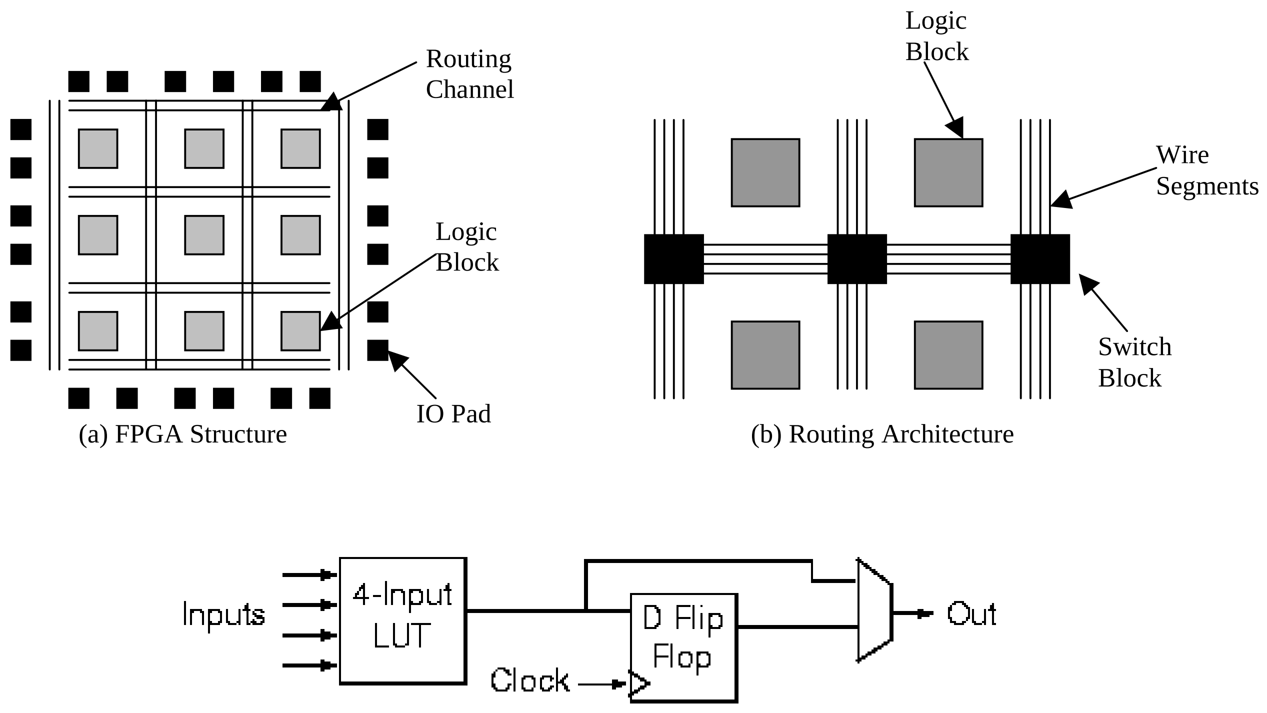
The objective of timing-driven placement is to place a set of logic blocks on the FPGA array structure such that the maximum delay of the circuit is minimised, satisfying timing criteria on the nets. We restrict our attention to island-style FPGAs, though the method proposed can be easily extended to row-based structures as well. The FPGA consists of an array of logic blocks (LBs) and routing channels. Each logic block has 4 inputs and an optional clock. In addition, each LB has two equivalent output pins, and both of them are available for routing. Each channel is assumed to have a fixed number of tracks. Also, the channels are assumed to be unsegmented, each segment spanning the distance of one logic block. Two IO pads can fit in the space of each LB along the periphery of the chip. Fig. 1 shows the architecture of the FPGA considered.

One of the major constraints in incorporating timing information at the placement stage is the need for accurate delay estimates. Delay estimation in FPGAs has been addressed in [3]. Traditionally, simulated annealing has gained importance as a method that can effectively use such reasonable estimates for constrained optimisation. Genetic Algorithms (GAs) and Evolutionary Programming are other popular methods that help in optimisation in a multimodal landscape [2,8]. The main difference between GA and EP is that EP does not rely on the crossover operator for evolving solutions [2]. GAs and EP operate on a family of potential solutions to determine the best of the available solutions, and continuously improve them. This search method is directed, and is known to be robust in locating the global optimum of a multimodal function. Evolutionary approaches to placement have been addressed in [1,4].

Since multiple iterations (generations) are involved in the optimisation process, usually the computation of exact delays and timing-slacks is avoided, and fast estimates are used. We use a net-based approach, as a path-based approach would be computationally costly in iterative methods. Details of the algorithm are provided in the ensuing section.

3. EVOLUTIONARY PLACEMENT:

Each member of the population is indicative of a particular solution to the problem, which is evaluated. The encoding of the potential solutions into such member strings, and the definition of the evolution strategy characterise the evolutionary programming method. During each generation, a large number of such member strings are evaluated for suitability, and the best members of the



(a) FPGA Structure (b) Routing Architecture

(c) Logic Block Architecture

Fig. 1. Architecture of the FPGA Considered

population are carried over to the next generation using a selection policy. The resemblance of the evolutionary programming approach to the natural process of evolution is complete with a suitable mutation mechanism, wherein small random changes are made in the evolutionary environment. The evolutionary environment consists of the population and the conditions under which the members of the population operate. Small changes in the environment help in getting out of local minima in a multimodal setup, so that convergence is only towards the global minimum.

3.1. The Population:

As mentioned earlier, each member is a string that has a solution coded into it. A direct method of encoding a solution is to use the (x,y) co-ordinates of the blocks in the FPGA array that are occupied by the blocks to be placed. However, this method would be heavily demanding on memory, as a large number of such solutions are considered during each generation. To overcome this constraint, we propose a relative encoding of the positions. During any particular iteration, a base (reference) placement is considered for all members of that generation. The initial base placement is a random one, and is progressively evolved over generations. During each generation, each logic block is allowed to move to one of its adjacent (surrounding) positions only. Each member string is a binary string. Binary strings are known to be efficient in representing the search space and are easy to be handled by the evolutionary operators [2,8]. The length of the string is equal to the number of blocks to be placed, and each bit is associated with a particular LB. Each bit indicates whether the corresponding block is to be moved from its present base position or not. If it is to be moved, a new vacant position from among the adjacent surrounding blocks is chosen randomly. If no moves are possible, the blocks are retained in their original positions. A small array is used to hold the new position assigned for each block, corresponding to the fittest member of the population of the present generation. The values in this array are incorporated into the base position for the subsequent generation. The population is evolved across generations using a selection operator, and mutation.

3.2. Fitness Function:

The fitness function is a function of the goodness of each member. We have incorporated the routing delay and the timing information of the nets as well as the congestion factor in evaluating the overall fitness function [3]. The dimensions of the bounding box for each net and the congestion of the bounded area are used to compute the delay estimate for each net. Initially, this information for all the nets is stored, corresponding to the base placement. This pre-computation reduces the computation required during each iteration because the delays of only those nets whose associated blocks are moved have to be recomputed.

Let t_n denote the delay computed for each net n, and the maximum allowed delay be T_n. The fitness function provides a credit to nets that satisfy timing constraints and imposes a penalty on those that violate timing requirements. The fitness function is of the form,

$$\mathbf{F(P) = A/(B + C(P))},$$

where, A and B are constants, P is a placement solution and C(P) is the cost parameter.

$$\mathbf{C(P) = \Sigma\, \Delta t_i (P)}\ ,$$

where $\Delta t_i (P)$ is defined as,

$$\mathbf{\Delta t_i (P) = \begin{cases} (t_i (P) - T_i) * PENALTY / T_i & , \text{if } t_i \geq T_i \\ (T_i - t_i (P)) * CREDIT / T_i & , \text{if } t_i < T_i \end{cases}}$$

The penalty factor is chosen to be much larger than the credit factor, so that violations are penalised greatly. Thus, in a given generation, high fitness strings are those whose corresponding placements have less violations.

The values of the constants depend on the problem instance under consideration. The results we present are for A=1000, $10 \leq B \leq 20$. The penalty and credit factors differ by a factor of 5-10, and credit is roughly equal to B/10. These values have been found to be suitable to the benchmark circuits we have worked with.

3.3. The Operators - Selection and Mutation:

The selection process we follow is one of proportional representation. Only the best strings are carried on to the next generation, and the number of instances of representation that each string receives in the next generation depends on the fitness value. Thus, highly fit strings replicate and find their way into the next generation. Since the strings in our method are bonded with the base placement, to maintain the direction of movement that brought about a better placement, the value of the new position is also copied to the next generation for each member. However, if a string is to be replicated, the additional copies get the usual random movement assignment. Here again, to conserve memory, only the movement to the adjacent blocks is coded as (-1,0,1) in both x and y directions.

```
Procedure EvolutionaryPlace {
        Define random binary strings for the population;
        Define random initial placement;
        Generation := 0;
        While (Generation < MAX_GENERATION ) do {
                Precompute fitness of base placement;
                For each member of the population do
                        Compute fitness;
                        Compute the maximum fitness of
                                the generation;
                Change base placement;
                Select strings;
                Mutate strings;
                Move and Swap blocks;
                Generation := Generation+1;
        }
}
```

Fig. 2. The Evolutionary Placement Algorithm

The process of mutation plays a very crucial role in the operation of the algorithm. Usually, mutation involves randomly changing member strings at arbitrary positions, with a small probability, pMut, so that solutions are not stuck up at local minima as a result of the proportional selection scheme. However, this level of mutation is insufficient for our method of evolutionary programming, as a result of the relative position encoding used. Such a mutation would result in only very small disruptions in the evolutionary environment that would be overshadowed by the proximity to a local minimum. Hence, we define two other operators operating on the other part of the environment, namely the base placement on which the members of the population are defined. These operators are the moves and the swaps of LBs. During each generation, a small percentage of blocks (pMov) are moved away from their present position to other vacant blocks. Also, the positions of a small percentage of pairs of blocks (pSwp) are swapped. The selection of blocks is at random. The number of blocks moved and swapped is progressively reduced over generations, so that better solutions are more likely to be preserved.

The overall placement algorithm is listed in Fig. 2.

4. RESULTS:

We have implemented the proposed algorithm in C++ in a Linux environment. We have tested the algorithm on the MCNC benchmark suite. We have considered various values of the mutation probability pMut, and found experimentally that for the benchmark circuits we have used, values between 0.03 and 0.08 give good results. Initially, about 2% of the blocks were moved and 2% of the blocks were considered for swaps. This number was progressively decreased with evolution. The population size we assumed was between 2 to 4 times the number of blocks to be placed.

Our evolutionary placement algorithm was used to place the logic blocks, while we used a simulated annealing schedule to place the IO pads along the periphery of the FPGA chip. The parameters used for EP for the benchmark circuits are listed in Table-1.

The input to the placement algorithm was a netlist after SIS optimisation and technology mapping. The SIS script used for this purpose is given in Fig. 3.

```
xl_part_coll -m -g 2 -n 4
xl_coll_ck -n 4
xl_partition -m -n 4
simplify
xl_imp -n 4
speed_up
xl_rl -n 4
xl_partition -t -n 4
xl_cover -e 30 -u 200 -n 4
xl_coll_ck -n 4
```

Fig. 3. SIS Optimisation and Technology Mapping Script

Circuit	No. of Blocks	No. of IO Pads	No. of Nets	Array Size	Population Size	No. of Generations	pMut*	PSwp*	pMov*
5xp1	47	17	54	10 x 10	175	15	0.05	0.02	0.03
b9	61	62	102	10 x 10	200	30	0.07	0.02	0.02
Cc	37	41	58	12 x 12	200	30	0.05	0.02	0.02
Clip	119	14	128	12 x 12	250	15	0.04	0.02	0.02
Comp	52	35	84	10 x 10	150	15	0.05	0.02	0.02
Count	93	51	128	12 x 12	150	15	0.04	0.02	0.02
f51m	47	16	55	10 x 10	150	15	0.04	0.02	0.02
Lal	61	45	87	10 x 10	150	15	0.04	0.02	0.02
Ldd	50	28	59	10 x 10	150	25	0.04	0.02	0.02
Pcler8	55	44	82	10 x 10	150	20	0.04	0.02	0.02

***: Initial values.**

Table - 1. Circuit and Algorithm Details.

Circuit	By Evolutionary Programming (EP), ns	By Simulated Annealing (SA), ns	By SA – EP, ns
5xp1	34.21	35.87	31.22
b9	37.07	35.87	33.22
cc	28.47	33.47	30.85
clip	57.48	55.26	49.85
comp	49.49	55.79	48.10
count	46.59	51.06	45.68
f51m	55.61	73.81	58.49
lal	32.08	39.74	35.68
ldd	38.55	36.22	33.78
pcler8	32.08	42.16	38.55

Table – 2. Total Net Delay for the Critical Path.

We used VPR, a place and route tool from the University of Toronto to evaluate the final delay after timing-driven routing. VPR can be used for placement based on simulated annealing, or can be used for routing on a given placement. This tool was used on the placement generated for routing and timing evaluation. The results obtained for our method and that for the simulated annealing method are shown in Table-2. Our method clearly yields better results compared to conventional simulated annealing based methods.

Better results were obtained on some of the circuits when we used our evolutionary placement approach on the results of the simulated annealing method. During this experiment, the moves and swaps were either disabled or were kept very small (0.1%). These better results, we attribute to the fact that by disabling the moves and swaps of blocks, the search-space of the evolutionary approach is limited to the blocks surrounding any particular block. Since the result of the SA is already near the global optimum, the EP approach hastens the convergence by a directed search, compared to the undirected random search of SA. These results are given in Table-2.

5. CONCLUSIONS AND FUTURE WORK:

The novelty of the timing-driven placement solution presented in this paper is the use of new methods of encoding and mutation that characterise the evolutionary programming approach used. The method proposed uses very less memory space compared to traditional methods of evolutionary solutions. The results obtained are encouraging and the algorithm outperforms conventional simulated annealing methods with respect to the results obtained.

Since evolutionary approaches can work fairly easily with multiple constraints, the proposed method can be extended to optimise other performance bottlenecks like power dissipation and signal skew. We are currently working in this direction.

Acknowledgments:

We are thankful to the Collaborative Benchmark Laboratories, North Carolina State University, for providing us with the MCNC logic synthesis benchmark suite. We are also thankful to Prof. Jonathan Rose and his group at the University of Toronto for having provided us with VPR, their FPGA placement and routing tool.

References:

[1] J.P. Cohoon and W.D. Paris, “Genetic Placement”, IEEE Trans. Comp. Aided Design, Vol.-6, No.-11, Nov. 1987, pp 956-964.

[2] D.E. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison-Wesley, Reading, Mass., 1989.

[3] Tanay Karnik and Sung-Mo Kang, “An Empirical Model for Accurate Estimation of Routing Delay in FPGAs”, Proc. Intl. Conf. Comp. Aided Design, 1995, pp 328-331.

[4] R.M. Kling and P. Banerjee, “ESP : Placement by Simulated Evolution”, IEEE Trans. Comp. Aided Design, Vol.-8, No.-3, Mar. 1989, pp 245-256.

[5] Anmol Mathur and C.L. Liu, “Compression Relaxation – A New Approach to Timing-Driven Placement for Regular Architectures”, IEEE Trans. Comp. Aided Design, Vol.–16, No.–6, June 1997, pp 597-608.

[6] Sudip K. Nag and Rob A. Rutenbar, “Performance-Driven Simultaneous Placement and Routing for FPGAs”, IEEE Trans. Comp. Aided Design, Vol.-17, No.-6, June 1998, pp 499-518.

[7] Srilata Raman, C.L. Liu and L.G. Jones, “A Delay Driven FPGA Placement Algorithm”, Proc. European Design Automation Conf., 1994, pp 277-281.

[8] M. Srinivas and Lalit M. Patnaik, “Genetic Algorithms – A Survey”, IEEE Computer, Vol.-27, No.-6, June 1994, pp 17-26.

[9] William Swartz and Carl Sechen, “Timing Driven Placement for Large Standard Cell Circuits”, Proc. Design Automation Conf., 1995, pp 211-215.