A GENETIC ALGORITHM FOR VLSI PHYSICAL DESIGN AUTOMATION*

Volker Schnecke, Oliver Vornberger
University of Osnabrück, Dept. of Math. / Computer Science
D–49069 Osnabrück, Germany

ABSTRACT

Solving discrete optimization problems with genetic algorithms is in many aspects different from the solution of continuous problems. The blindness of the algorithm during the search in the space of encodings must be abandoned, because this space is discrete and the search has to reach feasible points after the application of the gentic operators. This can be achieved by the use of a problem specific genotype encoding, and hybrid, knowledge based techniques, which support the algorithm during the creation of the initial individuals and the following optimization process. In this paper a genetic algorithm for the layout generation of VLSI-chips is presented, which optimizes two, usually consecutively solved tasks simultaneously: together with the placement of the modules, the routes for the interconnection nets are optimized.

INTRODUCTION

One of the main feature of a genetic algorithm applied to an optimization problem is the fact, that it does not deal with the problem itself, but with encodings of solutions for this problem. Thus the genetic algorithm explores the space of these encodings rather than the solution space itself. For continuous parameter optimization problems, both spaces are identically. A straight-forward genotype encoding in this case is a string of genes, which are simple floats. Each gene represents an element of the vector defining a point in the solution space. The standard mutation operator randomly modifies single genes, and crossover is done by direct merging of two gene strings, which results in two offspring. All offspring represent correct encodings and these encodings define admissible solutions to the given optimization problem, because of the one-to-one (genotype to phenotype) mapping between both spaces.

For discrete problems with string type genotype encoding, not every possible string represents a correct solution. It is even worse that simple crossing-over of two individuals does not necessarily lead to a correct offspring. There some repairment has to be done after crossing, which reduces the parent to offspring correlation.

For solving discrete real-world optimization problems, there has to be an application specific genotype encoding and 'intelligent' operators, which only create admissible individuals. During the application of these operators, problem specific knowledge can be used to generate high quality offspring. Here — in contrast to function optimization for example — bad genes, which would never be a building-block in a global optimal solution, could be recognized. The designer of a genetic algorithm can take care that these bad genes are not included in the population by hill-climbing strategies, which could be integrated in the construction of the initial individuals or during the application of the operators. In opposite to this, one could not know the good genes, i. e. the building-blocks which construct the optimal solution. Therefore, the designer has to support the genetic algorithm by presenting a pool of good genes. From this pool the algorithm can compose some good and (hopefully) eventually the optimal solution to the given optimization problem.

In the following, after a short description of the physical design process of VLSI-chips, a problem specific genetic algorithm for the layout generation is described. This approach takes into account the previous mentioned items by covering the following features:

- a problem specific genotype representation
- a hybrid approach for the creation of the initial population
- problem specific, 'intelligent' operators
- multiple gene representation in a single individual

LAYOUT GENERATION

Modern VLSI (very large scale integrated) microchips contain some million transistors. The design cycle for these chips consists of different serial steps (e. g. system specification, functional design, logic design, circuit design, physical design)[8]. The

*This work is being supported in the BMBF-project 'HYBRID–Applications of Parallel Genetic Algorithms for Combinatorial Optimization'.

physical design outlines the transformation of a circuit description (which is the result of the preceding circuit design process) into the *layout* of a chip, which is needed for the following fabrication step [3, 5]. The layout includes the geometric description of the circuit components and the information for the routes of the interconnections between them. It also has *pads* positioned on its borders for the I/O-connections of the chip. The objectives in layout generation are to minimize the area of the circumscribing rectangle and to produce a routing with short wirelengths, especially for some critical nets.

Due to its complexity, the physical design is usually divided into various, consecutive sub-steps: The circuit has to be *partitioned* to get a number of modules *(macro cells)* which have to be placed on the chip *(floorplanning)*. During placement there has to be enough space reserved to ensure the completion of all interconnections later on. In the *routing phase*, pins on the border of the modules have to be connected. This is done in two steps: In the *global routing* the 'loose' routes are determined, while in the *detailed routing* the exact routes for the nets in each channel between two modules are computed. The last step in the physical design is the *compaction* of the layout, where it is compressed in all dimensions so that the total area is reduced. The algorithm described in this paper combines the routing with the placement process during layout generation. For a more detailed description of the usual phases and possible solution methods in contrast to the approach described in this paper see [7].

THE INPUT

The input to the layout generation process for macro-cell layouts is a set of modules, which are rectangular blocks. Each module represents a functional unit which consists of hierarchically arranged subcells. There are two kinds of modules (cf. fig. 1): A *fixed module* has fixed dimensions with exact terminal positions for the interconnection nets on its borders (fig. 1, top). A *flexible module* can have various implementations with different aspect ratios, which are defined by a *shape-function.* This is a step-function which is characterized by a set of minimal width/height combinations (fig. 1, bottom). For the interconnection nets of a flexible module, only a list of terminals for each side is given but no exact terminal positions, because these vary with the different implementations.

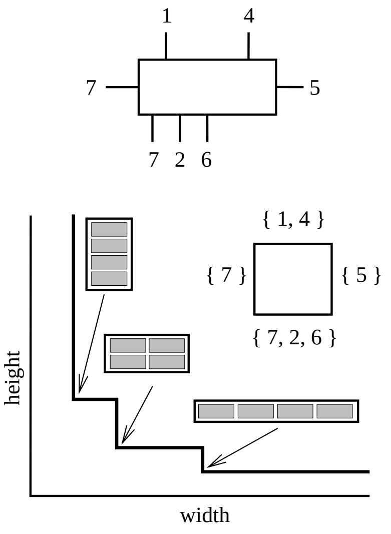

Figure 1: The input for a fixed cell (top) and a flexible cell with the shape-function (bottom)

A GENETIC ALGORITHM FOR LAYOUT GENERATION

A layout is described by the positions of the modules, the chosen implementation for the flexible modules and the routes of the interconnection nets on the layout-surface between the cells. Such a complex phenotype can not directly be represented by a gene-string of elementary data-types. A feasible way to characterize the placement of the modules is a binary slicing tree. This tree is the problem specific genotype encoding for the layout optimization (cf. fig. 2).

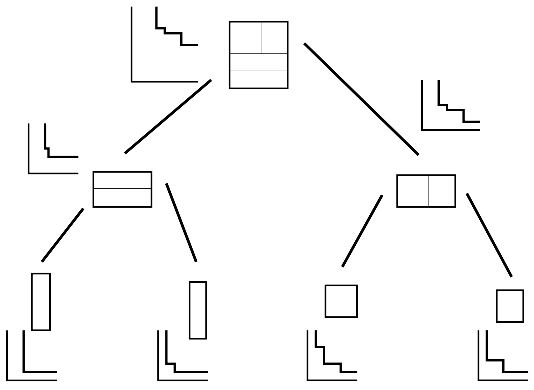

Figure 2: The genotype representation

The structure of the tree fixes the relative placement of the single *blocks* (modules) which are represented by its leaves. Each inner node represents a *meta-block*, which defines the arrangement for the set of blocks characterized by the leaves of the corresponding sub-tree and information about the routing inside this partial layout. All possible implementations for flexible blocks are taken into account by storing shape-functions for all nodes in the tree so that a single individual represents different layouts, if some blocks are flexible. When combining two blocks to a meta-block, the arrangement and the rotation of the composed blocks are fixed. Hence, the shape-functions of both blocks can be added which results in a shape-function for the meta-block. The number of implementations for the meta-blocks in the higher levels of the tree does not grow exponentially with their height in the tree because there are redundant implementations [7]. For the example shown in figure 3, two flexible blocks with three and two implementations are positioned upon. The shape-function for the resulting meta-block has only two different (non redundant) implementations. An upper bound for the routing space inside the meta-block is computed and added to its shape-function.

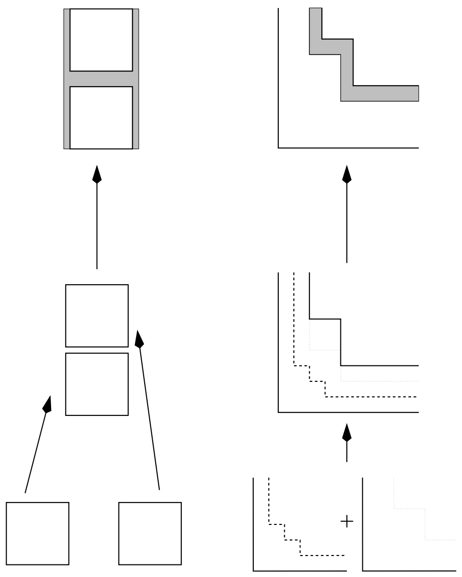

Figure 3: The combination of two flexible blocks to a meta-block and the addition of routing space

HYBRID CREATION OF INDIVIDUALS

The slicing-tree for an individual of the initial population is composed in a bottom-up fashion. A special heuristic – the *iterated matching* [2] – is used to create building-blocks which define high quality partial layouts. For that, a complete graph is constructed: the nodes represent the blocks, and each edge is weighted with a value which defines the quality of a meta-block consisting of the two blocks characterized by the adjacent nodes. A *matching* in this graph is a set of disjunct node pairs and the *maximum weighted matching* is the matching with the maximal sum of edge weights (cf. fig. 4). The quality of a meta-block is marked out by the number of common nets of the combined blocks.

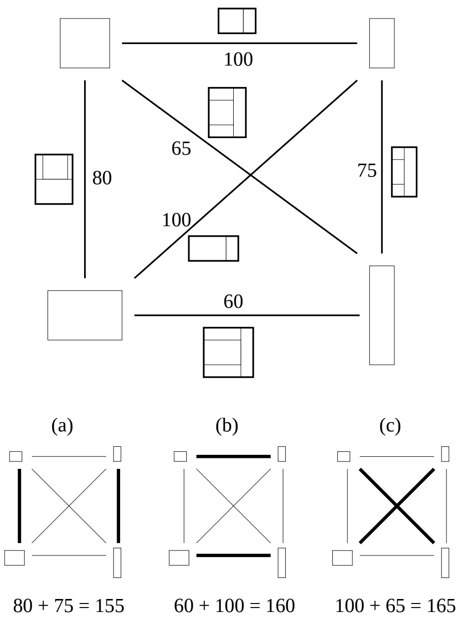

Figure 4: A matching graph for four blocks, the three possible matchings and the maximum weighted matching (c)

The matching process is iterated for each level of the tree until the root node is computed. In the second iteration, meta-blocks which consist of two blocks are paired, in the third iteration meta-blocks with four blocks are combined, and so on. This heuristic places highly connected blocks together and so reduces the overall wirelength and the total area of the layout. Because the iterated matching is a deterministic process, care has to be taken to create various individuals. For that, randomness is included in the computation of the edge weights for some of the used matching graphs.

THE INTEGRATED ROUTING

Because of the hierarchical construction of the in-

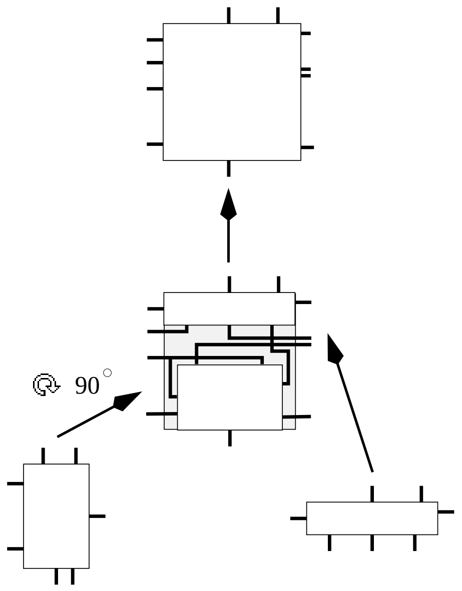

Figure 5: The construction of the detailed routing when combining a meta-block

dividuals and the binarity of the tree, the detailed routing on the layout can be computed during the placement of the blocks. Note that a meta-block is considered to be a fixed unit in the higher levels of the tree. When combining two blocks, all routing inside the resulting meta-block is done (cf. fig. 5). Terminals on the outer borders are passed on as terminals to the outer border of the meta-block. Terminals in the channel and on the outer sides of the blocks are connected, if they are shared by a common net. Nets inside the channel which could not be connected or have to be connected to more terminals than only those contained in this meta-block are passed on as terminals to one of the borders which are adjacent to the channel. For these nets, the direction of the way out of the channel is determined by a top-down traversal of the tree following the bottom-up construction phase. During this phase there is a check for each net, whether it is included inside of one of the partial layouts which are joined in an inner node. In this case, the net has to cross the channel and therefore a pseudo-optimal way for this net out of the channels in both sons of this node can be fixed (cf. fig. 6).

This method achieves that the nets follow the hierarchy of the cuts defined by the slicing tree structure. This way is often not optimal, but during the optimization process, the optimal structure of the slicing tree — regarding layout area and routing — is computed. In comparison to former experiments, where

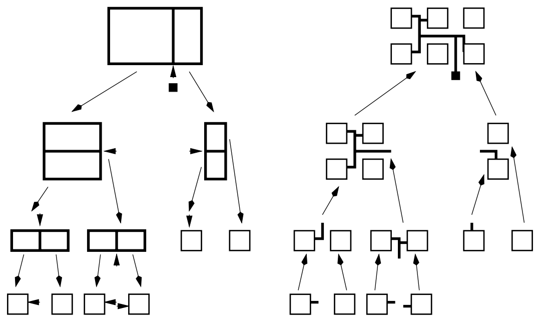

Figure 6: Fixing a global route by a top-down traversal (left) for the pseudo-optimal way (right)

the global routes were chosen randomly when combining the meta-blocks, here the total wirelength is considerably reduced. When choosing random routes for all nets out of the channels, many nets are routed to the outer border of the layout and have to be connected after composing the root node. Of course it could be possible to do the optimization of these routes by the genetic algorithm later on, but if one is able to compute (nearly) optimal routes 'by hand', there is no need for passing this work to the genetic algorithm.

MULTIPLE GENES

As mentioned before, all resulting implementations for the meta-blocks containing flexible blocks are stored. Storing all alternatives is useful because one could not decide in the lower levels of the tree, which implementation of a meta-block would be the best to minimize the overall area of the layout. Due to the simple adding of shape-functions and the binarity of the tree, the optimal sizing of the flexible modules to reach an optimal layout (optimal for the special placement) can directly be determined by a top-down traversal of the tree after fixing the best implementation for the root node [7].

This technique can be characterized as storing multiple genes in some locations of the genotype. It is a very good example for an opportunity the designer of a gentic algorithm has to increase its performance without directly guiding the search: when two blocks are combined, good combinations can be determined and bad combinations can be eliminated. But which of these good combinations are the best to be composed to a global optimal solution? Just let the genetic algorithm decide by making available a pool of good building-blocks!

THE OPTIMIZATION PROCESS

After the construction of the initial individuals, which already contain a lot of good components, the genetic algorithm starts the optimization by modifying individuals (mutation), and by combining building-blocks (crossover). Beside a mutation operator for changing the arrangement of two blocks inside a meta-block, the main mutation operators modify the slicing tree (cf. fig. 7). One operator exchanges blocks or meta-blocks, which corresponds to exchanging cells or partial layouts on the layout surface. The other important mutation operator changes the structure of the tree by randomly picking out a subtree and inserting it into the tree at a different position, which corresponds to moving cells or partial layouts on the layout surface. Here storing all important implementations for the meta-blocks once more enhances the performance of the genetic algorithm, because for a moved partial layout a different implementation might be better in its new environment.

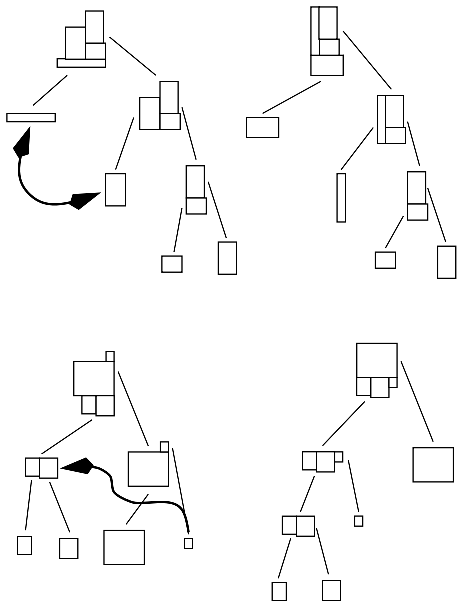

Figure 7: Mutation by exchaning blocks (top) and by changing the structure of slicing tree (bottom)

The implementation of the crossover operator is straight-forward: Two individuals are randomly chosen to produce one offspring. Two disjunct subtrees are searched in the parents which are composed to a subtree in the offspring. Because these subtrees usually do not build a complete layout, a third part has to be added to the layout of the resulting offspring. Due to this, the heritability (number of transmitted genes from the parents) is smaller than for problems where crossing-over directly leads to a correct individual. During the adding of the missing blocks into the tree of the offspring, the iterated matching is used once more to construct new (good) building-blocks. It has turned out to be not helpful to do some 'intelligent' crossover, i. e. looking for large disjunct subtrees, for example. Research is carried out to include a recombination operator for gene pool recombination. Here an offspring is constructed by combining building-blocks out of a pool of good partial solutions.

When designing a genetic algorithm for a specific problem, it is very important that a global optimum can be reached starting from any set of individuals by the application of the genetic operators. For an algorithm with tree-structured genotype encoding this means, if recombination only combines real subtrees with at least two leaves, then each pair of leaves contained in the encoding of an optimal solution must already exist in one of the initial individuals, or must be able to be constructed by mutation. Otherwise the genetic algorithm will never reach the optimum.

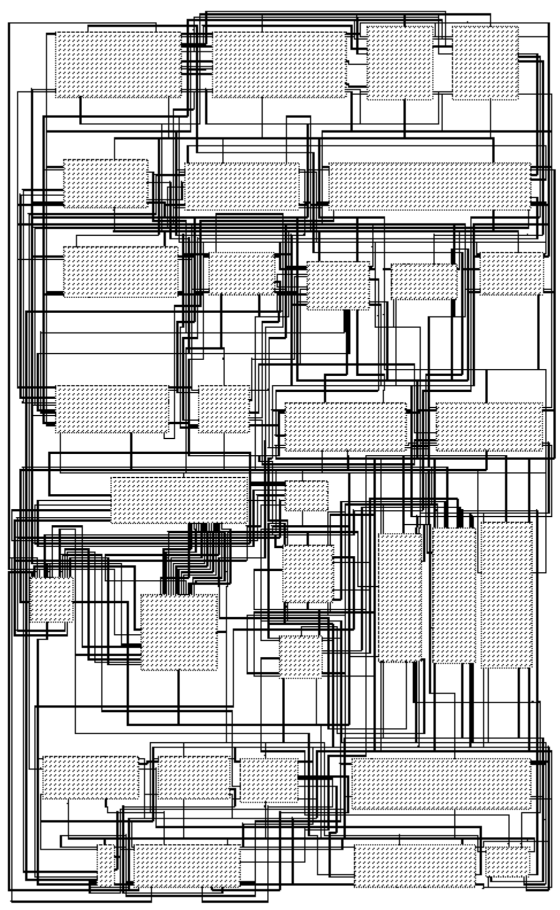

Figure 8: A layout with 33 macro-cells

RESULTS

The algorithm has been tested on real-world circuits with 10 to 49 modules and up to 500 nets. Figure 8 presents a layout for a circuit with 33 fixed modules and 123 nets. For a direct comparison to commercial placement and routing tools, an efficient channel routing algorithm has to be implemented. In the current version, a very simple routing strategy is applied when combining two blocks: for each net in the channel, one special track is added which results in an excessive routing space. Apart from the quantitative comparison it can be said that this is the only known approach to the layout generation process, which concurrently optimizes the placement together with the detailed routing.

FUTURE RESEARCH

A parallelization of the genetic algorithm is planned and along with this the implementation of a strategy adaptation [6]. There are many mutation operators, which are applied with different frequencies. It might be ingenious to exchange or move large parts of the layout during the early stage of optimization, and doing only minor changes when the population converges to an optimum. This can become possible by adapting the frequencies of the different mutation operators during the optimization.

Further a gene-pool recombination operator [1, 4] will be implemented which might replace the current crossover operator. For a combinatorial optimization problem like the layout generation, the biologically motivated crossing of two parent chromosomes is likely to be less efficient. The construction of an offspring out of a pool of good building-blocks seems to be more suitable.

CONCLUSIONS

For the application of genetic algorithms to optimize combinatorial problems, the focal point is to find a proper genotype encoding and all genetic operators, which are necessary to enable the algorithm to reach a global optimum. Solutions to real-world optimization problems are too complex for being represented as a simple gene-string. In the layout optimization process, due to the problem specific genotype encoding as a binary tree, the genetic algorithm is able to compute and optimize the routing on a chip concurrently with the placement of the modules. Usually in VLSI-CAD tools this is done in consecutive steps because of the complexity of the single optimization problems. But according to the strong interdependencies between the arrangement of the modules and the routing of the interconnection nets, it is wise to combine both steps.

The described application is a good example for the tasks which research on genetic algorithms should deal with: Because of the nondeterministic behaviour and the long runtimes, genetic algorithms will never succeed against other optimization methods for low complexity problems that allow fast greedy solutions. But for high complexity problems without any known sophisticated solution techniques, a genetic approach is well suited. By solving such a hard and complex problem, critics can easily be convinced of the power and the advantages of genetic algorithms. When designing a genetic algorithm for such a special application, it is important to withdraw from usual solution methods in one part, but using hybrid approaches and problem specific knowledge for hill-climbing in another. By doing so, there always is the trade-off between directed search and random search. The designer has to find out, which part of the optimization can be done 'by hand', how the genetic algorithm can be supported during the search process, and which tasks *must* be left for the genetic algorithm to work on.

REFERENCES

1. A. E. Eiben, P.-E. Raué, Z. Ruttkay, *Genetic algorithms with multi-parent recombination*, 3rd Conf. on Parallel Problem Solving from Nature, Springer Lecture Notes in Computer Science 866, 1994, 78–87
2. A. Fritsch, O. Vornberger, *Cutting Stock by Iterated Matching*, Operations Research Proceedings, Selected Papers of the Int. Conf. on OR 94, U. Derigs, A. Bachem, A. Drexl (eds), Springer Verlag, 1995, 92–97
3. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, 1990
4. H. Mühlenbein, H.-M. Voigt, *Gene Pool Recombination in Genetic Algorithms*, Procs. of the Metaheuristics Int. Conf., I. H. Osman, J. P. Kelly (eds.), Kluwer Academic Publishers, Norwell, 1995
5. S. M. Sait, H. Youssef, *VLSI Physical Design Automation: Theory and Practice*, McGraw-Hill (1995)
6. D. Schlierkamp-Voosen, H. Mühlenbein, *Strategy Adaptation by Competing Subpopulations*, 3rd Conf. on Parallel Problem Solving from Nature, Springer Lecture Notes in Computer Science 866, 1994, 199–208
7. V. Schnecke, O. Vornberger, *Genetic Design of VLSI-Layouts*, Procs. First IEE/IEEE Int. Conf. on GAs in Engineering Systems: Innovations and Applications, GALESIA '95, IEE Conference Publication No. 414, 1995, 430–435
8. N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, 1993