Genetic Scheduling Algorithm for High-Level Synthesis

P. Arató, Z. Á. Mann, A. Orbán
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
H-1117 Budapest, Magyar Tudósok Körútja 2.
Hungary
{arato, mann}@iit.bme.hu, Andras.Orban@cs.bme.hu

This paper was accepted for publication in the Proceedings of the
IEEE 6th International Conference on Intelligent Engineering Systems, Opatija, 2002

Abstract— **High-level synthesis (HLS) aims at constructing the optimal hardware or software structure from a given high-level specification. This process involves a number of optimization steps, from which scheduling is the most crucial one, concerning both the running time of the process and the quality of the found solution.**

In this paper, we present a genetic algorithm for the scheduling problem in HLS. We describe in detail how this general heuristic could be applied to the particular engineering problem at hand. It is demonstrated with several measurements on industry benchmarks that the new algorithm performs better than previous methods.

I. Introduction

The goal of High-Level Synthesis (HLS, [1], [2], [3]) is to automatically design the optimal hardware or software structure from the high-level (yet formal) specification of a system. A high-level specification may be for instance the description in a third-generation programming language such as C or some kind of pseudo-code. The optimality criteria may differ according to the particular application. In the case of hardware synthesis, the most important aspects are: the number of required processing units, chip size, heat dissipation, and energy consumption. Note that although HLS originally emerged from the field of hardware design, it raises similar problems as software design and synthesis. Therefore, similar methods may be used for hardware and software systems, and it becomes even possible to design hardware and software simultaneously (hardware-software co-design, system-level synthesis, [4]).

The main data structure of HLS is the elementary operation graph (EOG), which is an attributed data-flow graph. Its nodes represent elementary operations (EOs). An EO might be *e.g.* a simple addition but it might also be a complex function block. The edges of the EOG represent data flow—and consequently precedences—between the operations. The system is assumed to work synchronously; each operation has a duration (d), determined by its type, and a starting time (s). Therefore, node i works in the clock cycles from s_i to $s_i + d_i$. The correct operation of the system is defined by the following four axioms:

- Node j cannot start its operation until all nodes i, for which there is an $i \to j$ edge (*i.e.* the predecessors of j), have ended their operation;
- The inputs of node i must be constant during the total time of its operation (d_i);
- Node i may change its output during the total time of its operation (d_i);
- The output of node i remains constant from the end of its operation until its next invocation.

The process of HLS starts by building the EOG from the specification, then some transformations are made on the graph in order to meet timing and consistency requirements. Finally, the elementary operations are scheduled—*i.e.* their starting times are determined—and allocated in real processing units. The actual optimization is carried out in these last two steps. Both are computationally hard, but the big difference is that while there exist some quite effective approximation algorithms for allocation, this is not the case with scheduling. The output of the whole HLS process is the set of structural and control information that is needed to implement the resulting system.

At the Department of Control Engineering and Information Technology at Budapest University of Technology and Economics, a software called PIPE [2] has been developed, which realizes the HLS procedure. The unique feature of PIPE which distinguishes it from other HLS tools is that it can also handle pipeline systems, which are of great importance because pipeline processing can boost the performance of systems that are otherwise difficult to parallelize. A pipeline system is characterized by two numbers: latency (denoted by L) is the time needed to process one data item, while restart time (denoted by R) is the period of time before a new data item is introduced into the system. Generally $R \leq L$. Thus, non-pipeline systems can be regarded as a marginal case of pipeline systems, with $R = L$. If a large amount of data has to be processed, then minimizing R at the cost of a reasonable increase in L or hardware price is an important objective of HLS.

PIPE contains a so-called force-directed scheduler, which belongs to best known scheduling algorithms. Since scheduling is the most crucial phase of HLS, and the results of currently known scheduling algorithms are often not satisfying, new scheduling algorithms have to be invented.

In this paper a new genetic scheduling algorithm is presented. We describe in detail how this general heuristic could be applied to the scheduling problem of HLS. The results of the empirical tests on industrial benchmarks show that this new scheduler almost always produces better results (and often even in shorter running time) than the state-of-the-art force-directed scheduler.

The paper is organized as follows. Section II defines the scheduling problem and explains its most important

characteristics. Section III presents the new algorithm. In Section IV the empirical evaluation of the new algorithm is described. Section V concludes the paper.

II. The scheduling problem

A. Problem definition

The input of scheduling consists of: the restart time R; the latency L; and an EOG, in which each node is associated with an operation type that also determines its duration, as well as with an ASAP (As Soon As Possible) and ALAP (As Late As Possible) value for its starting time. (The [ASAP,ALAP] interval is often referred to as the mobility domain of the node.) *The task of the scheduler is to find an optimal valid scheduling in the given EOG between ASAP and ALAP.* The output is the vector of the determined starting times.

In this definition, the terms 'optimal' and 'valid' need further clarification. Valid means in this context that the precedence rules defined by the EOG are not violated. It is guaranteed that the ASAP and ALAP schedules are valid, but this does not necessarily hold for all schedulings between them. Therefore, the precedences must be assured explicitly.

Optimality should be measured in terms of hardware costs, *i.e.* the number of necessary processing units. However, it is known to be an $\mathcal{NP}$-hard problem to calculate the number of processing units for a given schedule (this is exactly the task of allocation) [2].

Therefore, the chosen objective is the number of compatible pairs instead. A pair of nodes is called compatible if they may be allocated in the same processing units; otherwise they are called concurrent. If the number of compatible pairs is high, this usually results in a low number of necessary processing units. Calculation of the number of compatible pairs is not straight-forward, since—as a consequence of pipeline operation—also nodes that are far from each other in the EOG can become concurrent. However, the Concheck algorithm [2] can determine the compatibility of two EOs in $\mathcal{O}(1)$ steps. This is why the number of compatible pairs was chosen as objective instead of the number of processing units.

This completes the definition of the scheduling problem. This slightly modified problem is also $\mathcal{NP}$-hard, which motivates the application of heuristic algorithms.

B. Known scheduling algorithms

There are three major groups of scheduling algorithms: those built on integer linear programming (ILP), list schedulers and force-directed schedulers.

In ILP-based schedulers, the scheduling problem is converted into a system of integer inequalities. The optimum of some linear function can then be found using standard linear programming techniques, *e.g.* the simplex method [5]. There are two major drawbacks to this approach. First, it is hard to give an estimate on the running time of the algorithm. Second, domain-specific knowledge cannot be built into the algorithm.

List schedulers are usually simple and fast because they process only once the time domain or the set of operations, trying to guarantee the optimal number of operations for each time cycle. This means a series of local, greedy decisions. Usually, some heuristic is used to define the order in which operations are selected. The problem with this approach is that a bad choice cannot be undone later. Thus, the result given by list schedulers is usually not good enough. ALso, the performance of list schedulers depends significantly on the order of the nodes.

The force-directed scheduler [6] tries to schedule approximately the same number of concurrent nodes for each time cycle, using a probabilistic approach. It is called force-directed because it always makes modifications proportional to the deviation from the optimum, resembling the law of Hooke in mechanics. This seems to be the best known scheduling algorithm, but its running time is proportional to the third power of the size of the input graph, so it can be slow for big EOGs. The performance of force-directed schedulers also depends heavily on the order of the nodes.

PIPE contains a force-directed scheduler. Since the actual optimization is done at this level of the HLS process, it is the most critical component concerning both the running time of the process and the quality of the found solution. For big input graphs, the running time of PIPE is essentially the same as that of the scheduler. This motivates the search for better and better scheduling algorithms.

III. Genetic scheduler

For the given $\mathcal{NP}$-hard optimization problem, we needed a general, high-performance search heuristic. This lead us to genetic algorithms (GA).

A. GA in general

Since there are very good surveys on genetic algorithms (see, for instance, [7], [8] and references therein), we will only give here a very brief, rather practical introduction only.

In order to implement a genetic algorithm, one has to do the following. First, an initial population must be set up somehow. After that, in each iteration a new population is generated from the previous one using the genetic operations: recombination, mutation and selection. So in each step there are two populations. The new population is first partially filled using recombination (usually there is a predefined recombination rate), then the rest using selection. Mutation is then used on some individuals of the new population (their number is defined by the mutation rate). The order of the operations is important because this way individuals created by recombination and mutation surely get into the new population, even if they are not very fit, so they have the chance to propagate their possibly good properties.

Mutation is important because it guarantees variety and thus helps leaving local optima. Its implementation is simple: a randomly chosen gene of a randomly chosen individual is altered randomly. Of course there can be variants:

one can alter one gene at a time or maybe more genes at once, one can allow only slight modifications of a gene or any modification etc.

The aim of recombination is to mix good properties of the individuals so that a sequence of genes corresponding to a good property can spread across the whole population. It is implemented by generating one or more (usually two) new individuals by the cross-over of two individuals. Individuals are usually not chosen with uniform probability for recombination (as for mutation for instance), since it is more probable to get good genes from individuals having high fitness values.

Selection is also necessary, it is used to propagate good individuals and to eliminate bad ones. It is implemented by simply copying the best individuals into the new population. In some variants it is not necessarily the best ones who survive, but they are selected with a higher probability.

B. *Application for the scheduling problem*

In this section we present the way that the above general scheme could be used for the scheduling problem. That is, we describe what are individuals, the population, genetic operations and the fitness function.

B.1 Individual

Actually, the scheduling problem is fortunate from the point of view of a genetic algorithm. The applicability of genetic algorithms requires that the solutions of the optimization problem can be represented by means of a vector with meaningful components: this is the condition for the recombination to work on the actual features of a solution.

Fortunately, there is an obvious vector representation in the case of the scheduling problem: genes are the starting times of the elementary operations. The order of the genes is not indifferent either: for the efficiency of recombination it is vital that genes next to each other do represent correlative pieces of information. It can be assumed though that the elementary operation graph was specified in a logical order (*e.g.* top-down, from left to right), so that the order of the nodes is suitable for genetic representation.

B.2 Population

Choosing the population is not as straight-forward as was the case with individuals. The question to answer is whether non-valid schedulings should be permitted.

Since non-valid schedulings have no real physical meaning, it seems to be logical at first glance to work with valid schedulings only. Unfortunately, there are two major drawbacks to this approach. First, this may constrain efficiency severely. Namely, it may be possible to get from a valid individual to a much better valid individual by genetic operations through a couple of non-valid individuals, whereas it may not be possible (or perhaps only in hundreds of steps) to get to it through valid ones only. In such a case, if non-valid individuals are not permitted, one would hardly arrive to the good solution.

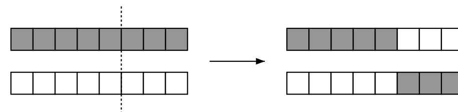

Fig. 1. Recombination of two individuals

The other problem is that it is hard to guarantee that genetic operations do not generate non-valid individuals even from valid ones. This holds for both mutation and recombination. Thus, if non-valid individuals are not permitted, the recombination operation cannot be used in the form of cross-over. Rather, it should be defined as averaging. But this method does not help to maintain variety in the population so it can cause degeneration. In the case of mutation it seems that the only way to guarantee validity is to immediately get rid of occasional invalid mutants. However, this would contradict the principle of giving every individual the possibility to propagate its characteristics.

For these reasons we decided to permit any individual in the population, not only valid ones. Of course the scheduler must produce a valid scheduling at the end. In order to guarantee this, there must be valid individuals in the initial population (see Section III-C.1) and the fitness function must be chosen in such a way that on one hand valid individuals do not become extinct, on the other hand it punishes invalidity (see Sections III-B.4 and III-C.3).

B.3 Genetic operations

Mutation, recombination and selection are used. Mutation is done in the new population; each individual is chosen with the same probability. Recombination is realized as cross-over: from two individuals of the old population two new individuals are generated as illustrated in Figure 1. The roulette method is used for choosing the individuals to recombinate. Selection is realized as filling some part of the new population with the best individuals of the old population. This is done by first sorting the individuals according to their fitness with QSort and then simply taking the first ones.

B.4 Fitness function

The fitness has two components: the first one is the actual objective function, namely the number of compatible pairs. If only valid individuals were allowed, the fitness would be equal to the objective function. But non-valid individuals are also allowed; however, they should have lower fitness values. This is why a second component of the fitness is needed. Since these individuals should be motivated to be less and less invalid, the second component of the fitness is a measure of the invalidity, namely the number of *collisions*, i. e. the number of precedence rules (edges of the EOG) that are corrupted. So the fitness is monotonously increasing in the number of compatible pairs and monotonously decreasing in the number of collisions. (For more details, see Section III-C.3.)

B.5 The output of the scheduler

Optimization can be made more efficient by means of a large population, but the scheduler must give only one solution at the end. However, there may be dozens of valid individuals with a high objective value in the last population. So we choose the best valid individuals and run the allocation process for all of them. Then the best one is chosen (in terms of used processors and not compatible pairs anymore) as output.

C. Implementation

In this section we take a closer look at our genetic implementation and describe some interesting details.

C.1 Initial population

In order to be sure that we get a valid scheduling at the end, some valid individuals must be placed into the initial population. (The fitness function will make sure that they will not be replaced by invalid ones.) It seems to be a good idea to have several valid individuals in the initial population so that computational power is not wasted on individuals with many collisions. Now the question is how to generate those valid individuals? Well, two valid schedulings are known in advance: ASAP and ALAP. It can be proven that any weighted average of two valid schedulings is also valid. This way, additional valid individuals can be generated.

In our program, the ratio of valid individuals in the initial population can be specified in the *sched.ini* file. Suppose that Z valid individuals are needed. Then individual i ($i = 0 \ldots Z-1$) has the form:

$$ASAP + (ALAP - ASAP) \cdot \frac{i}{Z-1}$$

Of course this method will not always generate Z different individuals. It has the advantage though that it is very simple and the generated individuals are homogeneously varied between the two extremes ASAP and ALAP. So it is likely that subsequent mutations and recombinations will generate very different valid individuals from these.

C.2 Roulette method

The aim of the roulette method is to choose an individual with a probability distribution proportional to the fitness values. It is realized as follows. Assume that the fitness is always positive and the individuals are ordered somehow, numbered from 0 to $n-1$. Denote the sum of the fitness values of the individuals $0 \ldots i-1$ by S_i. This means that $S_0 = 0$, S_n is the sum of the fitness values of all the individuals and S is monotonously increasing. Choose an arbitrary number $0 < m < S_n$. Suppose that m lies in the interval $[S_i, S_{i+1}]$. Then the chosen individual is the one numbered i.

Since the length of the $[S_i, S_{i+1}]$ interval is equal to the fitness of the individual number i, individuals are indeed chosen with probabilities proportional to their fitness. The method is called roulette because the intervals may be visualized on a roulette wheel, the roulette ball finishing in them with probabilities proportional to their sizes.

Building the S_i values requires $\mathcal{O}(n)$ time, but this has to be done only once in an iteration. The last step, namely finding the interval containing m, can be accelerated significantly as compared to the obvious linear search. Since the S_i values are monotonously increasing, binary search can be used, requiring only $\mathcal{O}(\log n)$ steps. Since cn individuals are chosen, the whole process requires $\mathcal{O}(n)+cn\mathcal{O}(\log n) = \mathcal{O}(n \log n)$ time.

C.3 Tuning the fitness function

The actual fitness function has great impact on the performance and efficiency of the system. As mentioned earlier (Section III-B.4), the fitness function has two components: the number of compatible pairs (call it *compat*) and the number of collisions (call it *coll*). After trying several functions, the following two proved suitable:

$$F_1 = maxcoll - coll + \frac{compat}{maxcompat}$$
$$F_2 = C_1 - \frac{1}{\frac{R}{L}\cdot(1+C_2\cdot coll)} \cdot coll + \frac{compat}{maxcompat}$$

In both cases, if the number of collisions is greater than a predefined $maxcoll$ value, the fitness is 0. Otherwise, the fitness is positive, which is necessary for the roulette method. (C_1 is set accordingly.)

The first function is based on the idea that the fitness function creates 'stairs' for the individuals, on which they can 'climb up' to better and better characteristics. There are small stairs and big stairs: decreasing the number of collisions corresponds to a big stair because it increases the fitness by 1. Increasing the number of compatible pairs corresponds to a small stair: it increases the fitness by $\frac{1}{maxcompat}$. This means that decreasing the number of collisions by 1 is worth more than any increase in the number of compatible pairs. This guarantees that the valid individuals do not die out and their number even increases.

The second solution is not so strict. In this case it is not claimed categorically that decreasing the number of collisions by one is worth more than any increase in the number of compatible pairs. Rather, a slight increase in the number of collisions is permitted if there is a sufficient increase in the number of compatible pairs. Now it must be defined what the 'sufficient increase' means. We found that this should depend on the value of $\frac{R}{L}$. Namely, if R is small compared to L, then there are lots of incompatible pairs and decreasing their number tends to increase the number of collisions. In order to avoid this, an increase in the number of collisions is only acceptable if there is a significant increase in the number of compatible pairs. On the other hand, if R is not much less than L, then it is not necessary to be that strict. Also, the increase in the number of compatible pairs that is needed to compensate an increase of 1 in the number of collisions should depend on the current number of collisions. For example, the increase of the number of collisions from 0 to 1 is much more costly than an increase from, say, 7 to 8.

C.4 Parameter handling

As can be seen from the above, our program has lots of parameters. This may be regarded as an advantage since it provides great flexibility. On the other hand, it makes evaluation and testing really hard. More on this in Section IV.

The most important parameters are the following: population size, number of iterations, recombination rate, mutation rate and the ratio of valid individuals in the initial population. There are also some non-genetic parameters, such as the name of the input file, R and L. There could be other parameters as well, *e.g.* bounds on where to cut the individuals in the case of recombination or the maximum change allowed in a mutation etc. But as described in the chapter about testing, evaluation was awkward enough without them.

Actually, the tests provided values for these parameters that seem to work well in most cases. So these are default values now but they can be overridden by means of command-line options or in the *sched.ini* file.

IV. Empirical results

At the beginning of the research our goal was to achieve better results than the existing force-directed scheduler of PIPE. Since PIPE consists of separable modules communicating only through text files, it is possible to replace the force-directed scheduler with ours without changing the code of other modules.

The algorithms have been tested on three benchmarks:

- Fast Fourier Transformation (FFT), 25 nodes
- IDEA cryptographic algorithm, 116 nodes
- RC6 cryptographic algorithm, 328 nodes

A. Aims of testing

The genetic algorithm can be configured with seven initial parameters. These settings have great influence on the efficiency and speed of the algorithm.

The user cannot be expected to know the internal parameters of the genetic scheduler, so we had to find the best combination of these parameters during testing and fix these values. Some parameters (like the size of the population, the number of steps) affect the solution in a more or less predictable way, while for others (recombination rate, mutation rate) we only have assumptions.

By increasing the size of the initial population we expect to find better scheduling, but of course the algorithm becomes slower. Allowing more steps provides significantly better results at first, but after reaching a certain point it has practically no more effect. By increasing the number of valid individuals in the initial population the algorithm does not have to 'waste power' to find valid solutions, but on the other hand, the initial population becomes less diversified, which can have negative influence on the quality of the result. To determine the appropriate recombination and mutation rate we can only use the trial and error way or rely on former experiences that can be found in the literature.

Problem	Force-Directed	Gen. v1	Gen. v2
FFT R=20 L=20	9	9	9
FFT R=20 L=30	11	6	-
IDEA R=100 L=268	17	15	15
IDEA R=100 L=278	16	15	16
IDEA R=200 L=268	13	10	11
IDEA R=268 L=268	6	6	6
IDEA R=278 L=278	8	7	7
IDEA R=50 L=268	25	25	25
IDEA R=50 L=278	29	23	23
RC6 R=10 L=201	210	207	207
RC6 R=10 L=211	23	15	15
RC6 R=100 L=201	25	23	23
RC6 R=201 L=201	13	11	11

TABLE I

The required number of processors

Another important task of testing is to decide which of the two objective functions produces better results.

B. Procedure of testing

Assuming that we only try 3 values for each parameter on the 3 benchmark problems with the two versions of the objective function, results in about 15000 executions. We used 4-8 computers simultaneously and thus we managed to complete the tests in two weeks. Most computers were working with Pentium II processors on Debian GNU/Linux platform. We implemented a TCL script to coordinate the test cases. The script has the following functions:

1. Runs PIPE with the given benchmark until the scheduling phase. This step provides the input for scheduling.
2. Runs the force-directed scheduler.
3. Runs the allocation on the output of the force-directed scheduler.
4. Runs both versions of the genetic scheduler with all the possible parameter-configurations.
5. Runs the allocation on each best individual of all the configurations of both versions. This means about 10^4-10^5 allocation runs for every benchmark!
6. Writes the duration and the required number of processors of every scheduling to a file in an appropriate format for later analysis.

C. Comparison of the results

The summary of the results concerning the number of required processors in the best configuration for each test case can be seen in Table I.

It can be seen that the new algorithms have reached the previous results in every case, moreover, in most tests they could improve them. This improvement is often remarkable, for example in the FFT $R = 20$, $L = 30$ test case the genetic algorithm could reduce the number of allocated processors to almost 50%. Apparently the genetic algorithm can cope with bigger tests as well, since it could lessen the required number of processors from 23 to 15 in the RC6 $R = 10$, $L = 211$ case. Another interesting observation is that by increasing the latency from 201 to 211 in the RC6

Fig. 2. The number of required processors of the two versions of the genetic scheduler in different configurations

$R = 10$ test we could reduce the number of processors to 10 percent of its previous value.

Figure 2 illustrates the influence of changing the parameters in the test case RC6 $R = 10$, $L = 211$ (the configurations are in the form: *size_of_population mutation_rate recombination_rate number_of_steps valid_rate*). Obviously, the two versions show quite different behavior, so selecting the appropriate objective function has the biggest effect on the quality of the result. Generally the first version had better results, but there were exceptions too.

Concerning only the number of steps, it can be derived from the diagram that after 100 steps we generally get the same results as after 300 or 500 steps. Another interesting observation is that increasing the size of the population from 150 to 300 leads sometimes to even worse results, however, the best result has been reached with a population of 300 individuals. It can be seen that changing the mutation and recombination rate affects rather irregularly the number of used processors. It is of course only a tiny fraction of all the parameter combinations, but it clearly illustrates the difficulty of configuring the genetic algorithm.

It is almost impossible to give the absolute best configuration, but some useful observations resulted from the tests. In the majority of test cases it was enough to allow 100 iterations. The size of the population should not be higher than 300. Producing 10 percent initial valid individuals also resulted in better figures. The best recombination rate was around $0.3 - 0.35$[1], the mutation rate was in the range of $0.1 - 0.2$.

It is important to note that in most practical cases the number of required processors is critical, not the running time. It is acceptable if the algorithm can finish in a couple of hours. Our algorithm is far below this limit.

Since the algorithms were implemented in different programming languages and tested on different machines, the running times can hardly be compared, but it can be stated that the genetic algorithm was surprisingly fast, in most cases faster than the force-directed scheduler. Moreover, we experienced that by stopping the genetic algorithm earlier, we can find a fairly good solution very fast, an order of magnitude faster than the force-directed scheduler.

[1] In our implementation this means that 60-70 percent of the new population will be generated by recombination.

V. Conclusion

In our research we focused on the most critical problem of HLS: scheduling. We presented in this paper a new genetic scheduling algorithm. The identification of individuals with schedulings was quite straight-forward; however, it was by far not obvious whether or not non-valid schedulings should also be allowed. We chose to allow non-valid schedulings as well, therefore, we had to make sure that the initial population contains valid schedulings and we had to choose the fitness function so that it encourages schedulings to be valid.

The empirical results on industry benchmarks show that this algorithm could effectively minimize the number of required processors and so it is able to reduce the cost of the planned system compared to the best previously known scheduler.

Currently, we are working on transferring these results of HLS to hardware-software co-synthesis.

VI. Acknowledgments

The work of the authors was supported by the grant OTKA T 030178 as well as a grant by Timber Hill LLC.

References

[1] D. Gajski., *High-Level Synthesis*, Kluwer Academic Publishers, 1992.

[2] Péter Arató, Tamás Visegrády, and István Jankovits, *High-Level Synthesis of Pipelined Datapaths*, John Wiley & Sons, Chichester, United Kingdom, first edition, 2001.

[3] R. Camposano, "From behaviour to structure: high-level synthesis," *IEEE Design and Test of Computers*, vol. 10, pp. 8–19, 1990.

[4] Ahmed A. Jerraya, M. Romdhani, C. Valderrama, Ph. Le Marrec, F. Hessel, G. Marchioro, and J. Daveau, "Models and languages for system-level specification and design," in *NATO ASI on System-Level Synthesis, Proceedings*, 1998.

[5] A. Schrijver, *Theory of linear and integer programming*, Wiley, 1998.

[6] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioural synthesis of ASICs," *IEEE Transations on Computer Aided Design*, 1989.

[7] L. Davis, *Handbook of genetic algorithms*, Van Nostran Reinhold, 1991.

[8] W. Kinnebrock, *Optimierung mit genetischen und selektiven Algorithmen*, Oldenburg, 1994.