High Level Synthesis with Genetic Algorithms - A Theoretical Approach -

Andreas Falkenberg
University of Hagen, Department of Computer Science
Institute of Theoretische Informatik
Feithstr 142, D-58093 Hagen (Germany)
Tel.: +49 (2331) 987-4784, Fax: +49 (2331) 987-339
E-mail: Andreas.Falkenberg@fernuni-hagen.de

ABSTRACT

Hardware synthesis is a very actual research theme in many institutes of computer science. The most algorithms used for hardware synthesis use single-criteria optimization. They improve the hardware either relating to speed or to area consumption. Genetic algorithms are easy to use in a parallel environment, like in a workstation cluster, and it is possible to define different multi-criteria cost functions. In this paper a graph theoretical approach is described, which is used in a synthesis system based on distributed genetic algorithms.

1. INTRODUCTION

The automated synthesis of hardware from an abstract description, for example from a VHDL-description [1] reaches more and more importance because of faster hardware design cycles. An algorithmic or behavioural description of a hardware module, whose correctness can be verified in a very early development phase using simulators, will be translated into a structural description by high-level synthesis tools. The aim is to find a solution with minimum time or area consumption. According to the user's priorities different algorithms to get fast or small hardware are chosen. One often used possibility is to calculate the minimum of the time consumption for a hardware module and according to this fixed solution to reduce the area to a minimum. On the other hand the upper bound of used modules can be fixed and according to this upper bound the time consumption is minimized. Because this problems are NP-complete [5], most of the algorithms do not guarantee the optimal solutions. A little difference of one parameter, for example the time consumption can make a big change to the other parameter. The most algorithms do not find a compromise, they try to find an optimal solution for exactly one parameter. Some algorithms allow the interactive input of constraints, but many trials are needed to find good solutions. By using genetic algorithms different solutions to a problem can be produced and rated by a cost function. Good solutions are selected by a selection-function, which searches for solutions with little cost. The advantage of genetic algorithms is the possibility to make good compromises. A weighted cost-function is used to choose the priorities whether a time or an area optimum or a good compromise shall be found. In this paper a weighted cost function is defined. The user can define his priorities using this cost-function and there is no need to repeat the synthesis with different constraints. An other advantage is the easy use of genetic algorithms in a parallel environment, for example a workstation cluster, so an acceleration is possible [2]. No or very few trials are needed to choose suiting constraints. The use of parallel genetic algorithms is effective if a number of workstations in a local area network is idle and can be used, for example at night.

2. THE SYNTHESIS PROCESS

The main steps of the synthesis process are the following:

1) Allocation of the module set BA

2) Assignment of the instructions to modules

3) Scheduling

In the first step a number of hardware-modules will be allocated. The second step assigns the elementary instructions to the allocated modules. The information of the assignment is used to schedule the instructions into time-steps. The time-consumption of every instruction can be calculated after it is known on which module the instruction will run.[6] The three synthesis steps are not independent to each other. So after many iterations and changes good solutions can be found. In the following chapter we use a fixed set of allocated modules and the assignment will be optimized by a genetic algorithm [7]. One result of the scheduling, the number of time-steps, will be used in the cost-function of the genetic algorithm. The optimization of the synthesis results by improving the allocated set of modules is presented in [4].

3. THE DEFINITION OF THE INSTRUCTIONS

The definition of the data-, anti-data, output- and self-dependencies are based on the following definition of the instructions.

Definition 1 *An elementary instruction is a 3-tuple* $(i, o, op) \in V^* \times V^* \times OP$. *With i is the tuple of input variables, o is the tuple of output variables and op is the operation of the instruction.*

We define the following functions with B is a set of instructions:

Definition 2 *The function* $f:B \rightarrow OP$ *returns the operation of an instruction.*

Definition 3 *The number of input- and output-variables of an operation are given by the functions* $\delta_{in}:OP \rightarrow N$ *and* $\delta_{out}:OP \rightarrow N$.

Definition 4 *The function* $in:B \rightarrow 2^V$ *returns the set of input variables of an instruction. The function* $in_j:B \rightarrow V$ *with* $j = 1 \ldots \delta_{in}(f(b))$ *returns the j-th input-variable of the instruction b.*

Definition 5 *The function* $out:B \rightarrow 2^V$ *returns the set of output-variables of an instruction. The function* $out_j:B \rightarrow V$ *with* $j = 1 \ldots \delta_{out}(f(b))$ *returns the*

j-th output-variable of an instruction b.

Lemma 6 It follows that $|in(b)| \leq \delta_{in}(b)$ *and*

$|out(b)| \leq \delta_{out}(b)$ *is valid.*

With this definition a compiler is needed to translate for example a VHDL-source into this format. In addition to this definition we add a label to some instructions. This label is a variable which represents the control-step the instruction is assigned to. This label will be set after the scheduling is done. We can define the instructions as follows:

Definition 7 With b is an elementary instruction and l is a variable, we can define a instruction as follows:

1) b is an instruction

2) l:b is an instruction

The examples show the translation of an if-statement and a loop-statement into the defined format. In Fig. 2 the instruction *while* assigns the variable *la*, which is a label to a variable *step*, which is the program counter. This translation into a data-flow structure intends, that the control-flow is trivial, because the control-instructions are included in the data-path.

if (u = 1)	*((a, b), (t), +)*
then	*((a, c), (e), -)*
x = a + b	
else	*((u, 1),(bed), =)*
x = a - c	*((bed, t, e),(x), if)*

Fig. 1: If-Statement as intermediate format

u = z + z	*1) ((z,z), (u), +)*
do	*2) la:((a,b), (x), +)*
	3) ((a,c), (y), -)
x = a + b	*4) ((b,c), (b), +)*
y = a - c	*5) ((u,x),(bed), <)*
b = b + c	*6) le:((la, step , bed),(step),while)*
while (u < x)	

Fig. 2: Do-while Statement as intermediate format

A program is given by a set of instructions and an order.

Definition 8 A program $PR = (b_1, \ldots, b_n)$ *defines an order of the set B. We write* $b_i <_{PR} b_j$ *if* $i < j$.

4. THE DEPENDENCIES

If b_i and b_j are two instructions, so that b_i is before b_j, we can define the following dependencies in the order of the program:

Definition 9 The data-dependency $da \subseteq \{(b_i, b_j) | b_i, b_j \in B\}$ *of two instructions* b_i *and* b_j *depending on PR is defined by:*

$(b_i, b_j) \in da$

$\Leftrightarrow (\exists l \exists k (out_l(b_i) = in_k(b_j)) \wedge (b_i <_{PR} b_j))$

$\Leftrightarrow (out(b_i) \cap in(b_j) \neq \varnothing \wedge (b_i <_{PR} b_j))$

b_j is called data-dependent on b_i (b_i da b_j) if and only if b_i writes to a variable which is needed by b_j.

Definition 10 The anti-data-dependency $ada \subseteq \{(b_i, b_j) | b_i, b_j \in B\}$ *of the instructions* b_i *and* b_j *in PR is defined as:*

$(b_i, b_j) \in ada$

$\Leftrightarrow \exists l \exists k (in_l(b_i) = out_k(b_j)) \wedge (b_i <_{PR} b_j)$

b_j is called anti-data-dependent on b_i (b_i ada b_j) if and only if b_i reads a variable which is written by b_j, but the data written by b_j are not those needed by b_i.

Definition 11 The output-dependency $aa \subseteq \{(b_i, b_j) | b_i, b_j \in B\}$ *of the instructions* b_i *and* b_j *in PR is defined as:*

$(b_i, b_j) \in aa$

$\Leftrightarrow \exists l \exists k (out_l(b_i) = out_k(b_j)) \wedge (b_i <_{PR} b_j)$

b_j is called output-dependent on b_i (b_i aa b_j) if and only if b_i and b_j write to the same variable.

The instructions are represented by the vertices of a data flow graph. Each of the three defined relations is represented by a directed edge between the corresponding vertices. A directed edge from b_i to b_j for example is constructed, if one of the relations (b_i *da* b_j), (b_i *ada* b_j) or (b_i *aa* b_j) fits. Thus, there is no difference in the construction of the graph between these three relations. For some algorithms it is important to know whether two instructions are indirectly dependent, this can be tested by the transitive closure of the relations.

Definition 12 The transitive closure T_R *of a relation R is defined recursive by :*

$(x, y) \in T_R \Leftrightarrow$

$(x, y) \in R \vee \exists z ((x, z) \in R \wedge (z, y) \in T_R)$

With this definition we are able to define a relation, which returns the indirect dependencies of two instructions. In Fig. 3 the graph for the example in Fig. 1 is shown. In this example only data-dependencies are used.

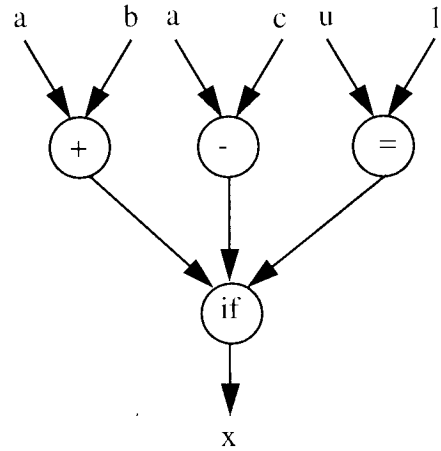

Fig. 3: If-Statement as data-flow graph

5. THE MODULES

After the definition of a module, the assignment process will be defined. We tried to give a very general definition of a module. By this definition it is possible to use multi-functional modules, like ALUs, and pipelined modules. Every operation of a module gets its own delay-time.

Definition 13 A module is defined as a 4n+1-tuple:

$$(o_0, \ldots, o_{n-1}, t_{10}, \ldots, t_{1(n-1)}, t_{20}, \ldots, t_{2(n-1)}, t_{30}, \ldots, t_{3(n-1)}, s) \in OP^n \times N^{3n} \times \Re$$

The first n elements o_i of the tuple are the operations of the module. The next elements t_{1i} define the delay-time of the operations o_i. The elements t_{2i} are important to define pipeline modules, they denote the latency time. If an operation o_i runs on the module then the next operation can be started after t_{2i}. The elements t_{3i} denote the delay-time in which the input of the module must be valid if the operation o_i is started. The time is not defined absolute, but as the number of time steps. The last element denotes the area consumption of the module. It is obvious that $t_{1i} \geq t_{2i} \geq t_{3i}$. In many cases $t_{1i} = t_{2i} = t_{3i}$ is valid.
If t_3 is known, the lifetimes of the variables can be reduced, which can lead to a lower number of registers. The knowledge of t_2 leads to faster designs. A set of modules *BLIB* is called library. If a library is given then we can define the allocation process, which develops a set of instances of modules *BA*.

Definition 14 The type of an instance of a module is returned by the function $bau: BA \rightarrow BLIB$.

The set of instantiated modules is a set of names, like the instantiation of variables in a program. The instances get their characteristics from the definition of the modules in the library. The following functions return the characteristics of an instance of a module. The information about the operations of a module is important.

Definition 15 The set of operations of an instance of a module is returned by: $f: BA \rightarrow 2^{OP}$ *with OP is the set of operations.*

The delaytimes of the operations will be returned by the following functions.

Definition 16 The delaytimes of a instance of a module are returned by: $T_1: BA \times OP \rightarrow N$,

$T_2: BA \times OP \rightarrow N$ *and* $T_3: BA \times OP \rightarrow N$.

The delaytimes of an instance of a module depend on the operation which runs on the module.

Definition 17 The area-consumption of an instance of a module is returned by: $r: BA \rightarrow \Re$

The module $u \in BA$ which is an instance of $(+, -, 2, 3, 1, 1, 1, 1, 7)$ is given. The functions defined above return for example the following results:

$f(u) = \{+, -\}$; $T_1(u, +) = 2$; $r(u) = 7$.

The optimization of the set of allocated modules is not a topic of this paper, so we suppose, that the allocation is optimal.

6. THE ASSIGNMENT

One solution of the assignment problem can be defined as:

Definition 18 An assignment of instructions to modules is given by a function $A: B \rightarrow BA$.

The data flow graph will be extended with non directed edges, so that every two vertices whose instructions are executed on the same module are connected by a non-directed edge. We can define a relation which is reflexive, symmetric and transitive:

Definition 19 With $b_1, b_2 \in B$ *and A is an assignment the following is valid:*

$b_1 \cong_A b_2 \Leftrightarrow A(b_1) = A(b_2)$.

The non directed edges have to be altered into directed edges, provided that no directed circles arise. If the direction of an non directed edge is chosen we can define an other relation:

Definition 20 The instruction b_j *is module-dependent on* b_i *if* $b_i \cong_A b_j$ *and* b_i *will be executed before* b_j *on the module u.*

With the definition of the order of the instructions assigned to the same module, we can define a set of valid permutations as follows:

Definition 21 An order of instructions is given by a permutation $p = (b_1, b_2, \ldots, b_n)$ *which is defined by the module dependency* $b_i <_u b_j \Leftrightarrow i < j$.

With this definition we can define the set of permutations, which are the possible orders of a set of instructions:

Definition 22 PE(u) is the set of orders, of the instructions, assigned to the module u.

To get the right direction of every edge, the instructions which run on the same module have to be sorted according to the partial order which is given by the union of the dependencies and the already sorted instructions.

Definition 23 With $u_1, \ldots, u_n \in BA$ *is the set of modules. An order* $p \in PE(u_i)$ *is valid if and only if*

$x <_{u_i} y \Rightarrow (y, x) \notin$

$T_{da \cup ada \cup aa \cup <_{u_0} \cup \ldots \cup <_{u_{i-1}} \cup <_{u_{i+1}} \cup \ldots \cup <_{u_n}}$

is valid.

The definition ensures, that the transitive closure of the union of the dependencies exists. If an order of two instructions is given by a data-dependency, the module-dependency can not contradict. Only valid assignments are allowed, so every instruction $b \in B$ can only be assigned to a module which is able to run the operation of the instruction.

Definition 24 An assignment $A: B \rightarrow BA$ *is valid exactly if* $\forall (b \in B)(f(b) \in f(A(b)))$ *is valid.*

The overloaded function f returns the operation of a instruction and on the other hand the set of operations of a module. Arithmetical logical units can be used which run different sets of operations.

7. THE GENETIC ALGORITHM

The main process of the genetic algorithm is shown in Fig. 4.

1)Generate a set of solutions $G_{as} \subseteq L(B, BA)$

2)Calculate cost $ges_{as}(A)$

3)Selection of solutions $Sel(G_{as}) \subseteq G_{as}$

4)Generate solutions $G'_{as} = gen(Sel(G_{as}))$

5)Start with G'_{as} at 2)

Fig. 4: Genetic algorithm for assignment

The genetic algorithm improves the assignment of the instructions in B to the modules in BA. Every instruction must be assigned to a module, which can run the instruction. The scheduling problem is a part of the costfunction and has to be solved independent for every assignment. The set $L(B,BA)$ is the set of all valid assignments which can be found for a pair B and BA. In the first step of the genetic algorithm a set of valid assignments will be generated. The cost of every assignment must be calculated with $ges_{as}(A)$. A selection of the assignments with the lowest cost will be used to generate new assignments with mutation and crossover functions. The coding, mutation, crossover and decoding functions are defined in the following section[3] :

Definition 25 A code for the assignment is defined by the function $C_{as}: L(B, BA) \rightarrow \Sigma^{|B|}$.

This means, every instruction is assigned to a symbol of an alphabet $\Sigma = N$. In our case this are the real numbers.

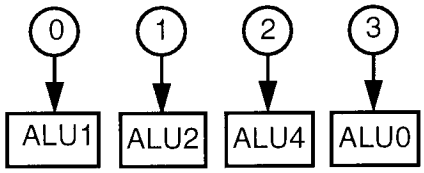

Fig. 5: Coding of the Assignment

Fig. 5 shows the coding of four instructions.

Definition 26 The mutation of an assignment is a function $m_{as}: \Sigma^{|B|} \rightarrow \Sigma^{|B|}$.

A mutation of the assignment is a function which changes the assignment of the instructions to the modules. Fig. 6 shows an example of a mutation. The mutation is a little change in the assignment, mainly only one module is changed. It must be explicitly ensured, that the new assignment is valid. A second mutation-operator changes the processing order of module-dependant instructions as illustrated in Fig. 7.

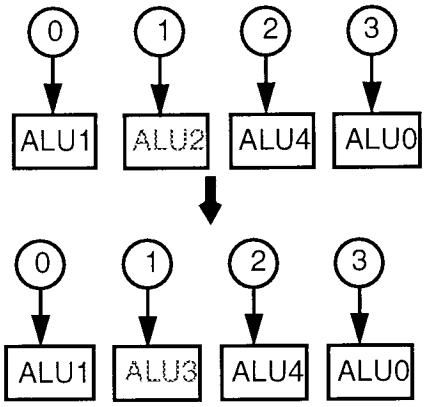

Fig. 6: Mutation eines Assignment

Definition 27 A mutation of the module-dependencies of a module u is given by a function $m_u: PE(u) \rightarrow PE(u)$.

The crossover operator, which mixes two assignments, is illustrated in Fig. 8 and works on the codes.

Definition 28 The crossover of the assignment is a function $cr_{as}: \Sigma^{|B|} \times \Sigma^{|B|} \rightarrow \Sigma^{|B|}$.

The crossover operator can not construct a invalid solution from

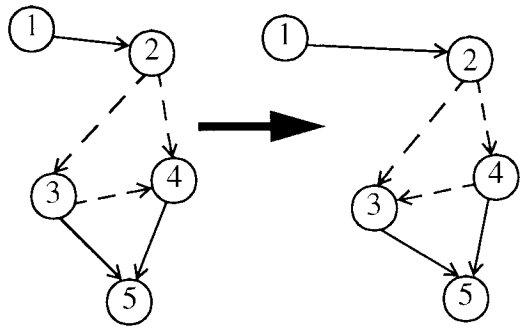

Fig. 7: Changing the processing order

two valid solutions. This is an important difference to the mutation operators, where an additional test must ensure that the solutions are valid. The instructions are represented by the vertices

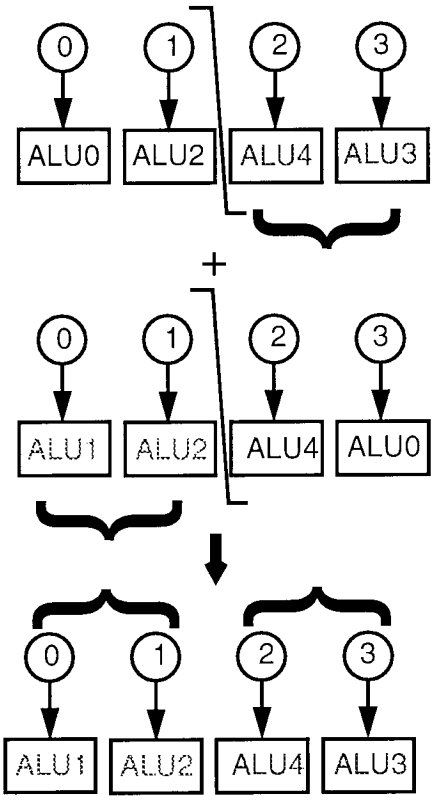

Fig. 8: Crossover of two assignments

of a data flow graph. Each of the three defined relations is represented by a directed edge between the corresponding vertices. A directed edge from b_i to b_j for example is constructed, if one of the relations (b_j *da* b_i), (b_j *ada* b_i) or (b_j *aa* b_i) fits. In addition to this three relations, the module dependency must be taken into consideration. Every dependency is represented by a directed edge. Further edges represent those instructions executed by the same module. The value of each edge can be calculated by the function k:

$$k(x,y) = max\left\{ k \;\middle|\; \begin{array}{c} k = T_1(x) \Leftarrow da(x,y) = 1 \\ k = T_2(x) \Leftarrow x <_u y \\ k = T_1(x) - T_1(y) + 1 \Leftarrow aa(x,y) = 1 \\ k = T_3(x) - T_1(y) \Leftarrow ada(x,y) = 1 \end{array} \right\}$$

In the ASAP algorithm shown in Fig. 9 not only the dependencies, but as well the delaytimes are taken into consideration. The function $sch(b)$ used by the algorithm is the starting time of the

instruction b. With this theoretical background on high level

```
Init:
for each node b
    sch(b) = 0
ASAP:
repeat
    for each node b_i
     for each node b_j
        if (it exists an edge from b_j to b_i)
            sch(b_i) :=
                max(sch(b_i),sch(b_j) + k(b_j,b_i))
     end for
    end for
until (no sch(b_i) has changed)
```

Fig. 9: Modified ASAP Algorithm

synthesis, other algorithms can be used as well. In the following section the costfunction or fitness is defined for the genetic algorithm.

8. THE COSTFUNCTION

The costfunction of the assignments can be defined as weighted sum of special cost functions:

Definition 29 The main costfunction of the assignment is defined as $ges_{as} = \sum_{i=1}^{6} w_{as,i} \cdot g_{as,i}$ *with the weightvector* $w_{as} = (w_{as,1}, \ldots, w_{as,6})$.

The cost of synthesis-solutions can be calculated with the costfunctions shown in Fig. 10. The number of time-steps of a synthesized circuit can be calculated by an ASAP (as soon as possible) scheduling, which takes care of the dependencies created by the assignment, as shown in the last section. The scheduling algorithm has to take care of the different time consumption of the instructions taken from the assigned modules.

$g_{as,1}$: Number of time-steps
$g_{as,2}$: Number of modules
$g_{as,3}$: Area of modules
$g_{as,4}$: Number of registers
$g_{as,5}$: Number of multiplexers
$g_{as,6}$: Area of wires

Fig. 10: Costfunctions

Definition 30 The number of time steps of an assignment A can be calculated by:

$$g_{as,1} = max\{sch(b_i) + T_1(b_i) | b_i \in B\}$$

The function *sch* assigns every instruction to a time step. T_1 returns the runtime of every instruction, which depends on the assigned module. The knowledge of the number of used modules is important if the area of every module is not known yet.

Definition 31 The number of used modules of the set BA by an assignment A can be calculated as:

$$g_{as,2} = |\{v | (v \in BA)(\exists b \in B)(v = A(b))\}|.$$

The area of the modules can be calculated if the area of each module is known.

Definition 32 The area of the used modules of the set BA by the assignment A can be calculated by:

$$g_{as,3} = \sum_{u \in \{v | (v \in BA)(\exists b \in B)(v = A(b))\}} r(u)$$

To calculate the amount of registers used for the synthesis of a circuit, we can assume, that for every variable one register is needed. After the calculation of the lifetimes of every variable they can be connected by a left-edge algorithm.

Definition 33 A register can be defined as set $re \subseteq V$ *of the set of variables. For all registers follows*

$re_j \neq re_k \Rightarrow re_j \cap re_k = \{\ \}$ *and*

$\bigcup_{i \leq n} re_i = V$, *with n is the number of registers.*

Using the left-edge algorithm a minimum number of registers can be calculated, given by $g_{as,4}$. The number of multiplexers is the sum of the multiplexers which are at the input of registers and of modules.

Definition 34 The number of multiplexers is calculated by the function $g_{as,5}$. *The input of a register re is connected to the output of a multiplexer if :*

1) $|re| > 1$ *or*

2) if v is the only element of re and it is used as output of more than one modules.

The inputs of a module $u \in BA$ *are connected to the output of multiplexers if* $A(u) > 1$.

Most synthesis systems do not take care of the area consumption of the wires, which can not be calculated exactly, before the final place and route. The defined function calculates an approximation of the area-consumption. For the calculation it is assumed that the length of a wire is proportional to the side-length of the chip, on a quadratic chip, it is the square route of the area.

Definition 35 Adding one wire to a chip with the area F_i *the new area* F_{i+1} *can be approximated as:*

$$F_{i+1} = F_i + c\sqrt{F_i}.$$

The number c is a constant which represent the expected area needed for one wire in relation to the existing area. The area F_0 is the amount of area for the modules, registers and multiplexers.

Definition 36 The expected area of n connections on a chip where F_0 *is the area needed by the modules, registers and multiplexers can be defined as:*

$$g_{as,6} = F_n - F_0.$$

The user finally has to choose the weight-vector $w_{as} = (w_{as,1}, \ldots, w_{as,6})$ according to his individual priorities. The multi-criteria optimization brings much benefit, because even a little change in the upper-bound of area can cause a great benefit in the time consumption. Especially the approximation of the area consumption of the connections is very important. Up to 90% of the area of a chip can be used by wires.

9. RESULTS

The first result is a comparison of a genetic algorithm to simulated annealing using the costfunctions. As shown in Fig. 11 simulated annealing is faster than the genetic algorithm on one workstation. But if a workstation cluster is available in a local area network, an acceleration of the genetic algorithm is possible as shown in Fig. 12. Especially if the power of workstations is available for example at night, the use of the genetic algorithm brings better solutions in a shorter time. The acceleration of the

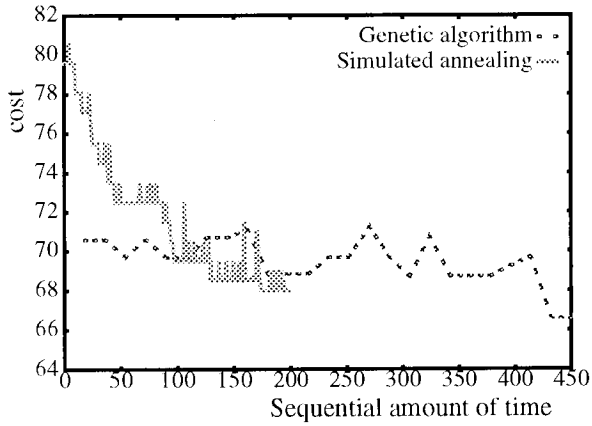

Fig. 11: Results

genetic algorithms is based on the parallel calculation of the costfunction for one generation, so acceleration is possible, until the number of workstations reaches the number of solutions in one generation. In Fig. 13 the results of a combination of the genetic algorithm with a heuristic methods are shown.They show that an acceleration of the synthesis process can be reached by the use of parallel genetic algorithms in a workstation cluster combined with heuristic methods. The presented theoretical approach is not reduced to genetic algorithms, but is a foundation to very different optimization algorithms for high level synthesis.

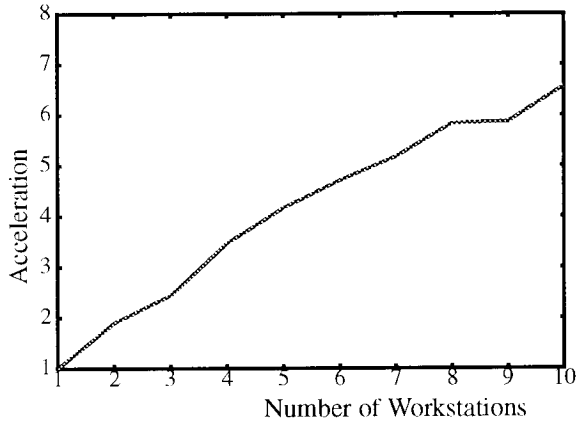

Fig. 12: Acceleration of the genetic algorithm

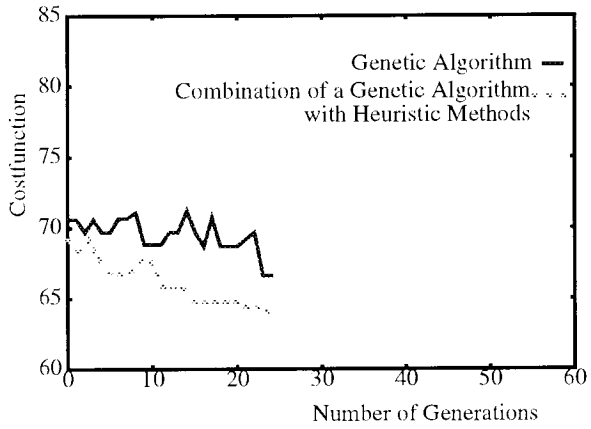

Fig. 13: Acceleration of Synthesis using optimized Allocation and additional heuristics.

10. CONCLUSION

In this paper different aims are described, on one hand the possibility of multi-criteria optimization of synthesis results, which are based on different costfunctions. A comparison between genetic algorithms and simulated annealing shows that the genetic algorithms is useful in a parallel environment, for example if a number of workstations is usable in a local area network, and are not used, for example a students workstation pool at night. Genetic algorithms can be accelerated nearly linear relating to the number of workstations. As shown in the last section mixing genetic algorithms with other methods, for example heuristic methods, leads to a further acceleration of the optimization process. In this paper we mainly presented a (graph) theoretical foundation, to the problem of high level synthesis. Using this theoretical approach algorithms can be developed, which use the presented benefits, which are a fine definition of the delaytimes, the module-dependencies and especially the genetic operators and cost functions.

11. REFERENCES

[1] Airiau R. ; Berge J.-M. ; Olive V. ; *Circuit Synthesis with VHDL*; Kluwer Academic Publishers 1994.

[2] Albert J. , Hinker S. , Schoof J. ; *Parallele Evolutionäre Algorithmen auf einem heterogenen Workstation-Cluster* ; SIWORK 1996.

[3] A. Falkenberg, F. Burchert, D. Tavangarian: "*A New Approach Towards Accelerating VLSI-Synthesis*", The Second World Conference on Integrated Design & Process Technology IPPS, Austin/Texas, 1996.

[4] Falkenberg A. : *Acceleration High Level Synthesis using fast Allocation* ; ICCIMA 98 ; Australia 1998.

[5] De Jong, K. ; Spears W.M. ; *Using Genetic Algorithms to Solve NP-Complete Problems*; Proceedings of the third International Conference on Genetic Algorithms; Morgan Kaufmann Publishers; Los Altos CA, 1989.

[6] Marwedel P. : *Introducing Complex Components into Architectural Synthesis* ;(ASP-DAC), Chiba, Japan, 1997.

[7] Shahid A. ; Sadiq M. S. ; Benten M. S.T. ; *GSA: Scheduling and Allocation using Genetic Algorithm* ; EURO-DAC '94 Grenoble France 1994.