

Grammar-Based Optimization of Synthesis Scenarios

Andreas Kuehlmann

Lukas P.P.P. van Ginneken

IBM Thomas J. Watson Research Center
Yorktown Heights, N.Y., U.S.A.

Abstract

Systems for multi-level logic optimization are usually based on a set of specialized, loosely-related transformations which work on a network representation. The sequence of transformations in a synthesis scenario (script) is crucial for the performance of the whole system. This paper presents the application of a genetic algorithm for automatic tuning of scenarios, and therefore an approach for optimizing the synthesis process itself. We introduce a general context-free grammar to describe the set of relevant scenarios. For such grammars we develop meaningful genetic operators for the implementation of an evolutionary search. Experiments with an industrial logic synthesis system show, that our method can improve the efficiency of manually designed standard scenarios by an average of 8%. We also show, that this approach can effectively tune scenarios to particular designs styles and/or specific synthesis goals.

1 Introduction

Although the problem of multi-level logic optimization can be adequately formalized, monolithic algorithms for a general purpose solution have not succeeded so far. Practical implementations of synthesis systems use a broad suite of special purpose algorithms which handle specific tasks towards a common synthesis objective [1, 2]. These tasks include steps for technology-independent optimization (e.g. constant propagation, redundancy removal [3], common subexpression sharing [4], Boolean minimizations [5]), technology-dependent optimization and mapping [6], and timing correction [7].

Typically, the individual algorithms are well understood. They are based on precise models which either guarantee an exact solution with respect to certain criteria or involve heuristics within a well defined scope. Further we will call such individual algorithms *transformations* (transforms). Most of the theoretical and practical work published in the logic synthesis domain

is related to such transformations.

In contrast, the relation between these different algorithms is less clear. Although their specific task in the general synthesis flow imposes a partial order, much freedom exists in sequencing them. For example, technology-independent optimization must be performed before technology mapping. However, within the technology-independent part, many different sequences of transformations may be applied. Moreover, it is essential to repeat certain algorithms in a different context or to iterate several times through whole subsequences in order to obtain best results.

Practical implementations of synthesis systems reflect this modular structure of underlying algorithms. For example, in LSS [1], the individual algorithms are realized as transformations working on a uniform network representation. About 100 technology-independent transforms and a number of technology-specific transformations are used. Options for the algorithms which can not be predetermined are left open as parameters. A *scenario* arranges a set of transforms and their actual parameters in a synthesis procedure. MIS [2] uses scripts (another expression for scenario) in a similar way. Scenarios and scripts are usually written in a programming language and are either interpreted at run time or precompiled.

The performance of most transforms strongly depends on the network structure at the time of their application. Therefore, it is crucial for the efficiency of a scenario to use the right order and a proper set of parameters. However, the best scenario depends on many conditions such as the initial design structure, the set of technology primitives, the synthesis goal in terms of size, speed, testability, and power consumption and on whether the design is control dominated or involves data flow.

There are two major problems involved in the development of good scenarios: First, general assumptions about design structures cannot be made, and second, the interactions between individual transforms are generally not well understood, are dependent on

the specific application, and are consequently difficult to predict. Therefore, scenarios are usually designed manually based on synthesis experience and many time consuming experiments [8]. Whenever existing transforms are modified, new ones are added or the technology changes, the process must be repeated to ensure that the scenario produces good results.

This paper proposes an algorithm for automatically generating and tuning logic synthesis scenarios based on a genetic algorithm [9, 10]. Genetic algorithms model natural evolution by simulating subsequent *generations* of *individuals* in a competitive environment. Starting from a randomly generated population, successive generations are created by two basic operators: (1) *crossover* which pairs two parent individuals to form two children and (2) *mutation* which applies a random small change to a single individual. The selection probability for parent individuals is weighted by their *fitness* which is derived from the search objective of the underlying problem.

Our results led to two important conclusions: (1) by applying the proposed tuning method the quality of synthesis scenarios designed by experts can be significantly improved, (2) an optimal general purpose scenario does not exist; by tuning the scenario to specific designs significant improvements can be obtained.

2 Optimization of Synthesis Scenarios

Results of manual tuning of MIS-II scripts were presented in [8]. The individual transformations in MIS-II were modified in order to improve the robustness of the system. As already mentioned, manual tuning of scenarios is time consuming and is problematic because of the lack of knowledge about the relation between individual transforms. In [11] a flexible method for MIS script optimization was presented. Based on a set of rules the most promising transformation is dynamically selected. Input parameters to the rules are the current network size, the sequence of transforms applied so far, and results from previous applications. The problem with this approach is that the complexity of the rule set is difficult to manage especially as the number of transformations and their possible interactions increases.

General approaches for optimization problems where little is known about the internal structure, are statistical search schemes like simulated annealing [12] or genetic algorithms [9, 10]. We favored the latter because of the following advantages:

- The search can be initialized with several given solutions. Instead of a single solution a whole population of different individuals with various val-

able properties is maintained. We use previously designed scenarios as a starting point which performs well with respect to given criteria.

- The effects of transforms are determined by their context. In many cases whole subsequences of transforms perform well because of their direct and immediate interactions. The implementation of mutation and crossover operators can reflect this characteristic by replacing or exchanging successive parts of the scenarios.
- The evaluation of the individuals for each population can be done in parallel on many machines. The number of machines which can be employed is bounded by the size of the population times the number of benchmarks for their evaluation.

A synthesis scenario can be regarded as a general program which we want to optimize with respect to a given criteria (area, delay). The scenario is usually written in a command language which syntax is well defined by a formal grammar.

3 Representation of the Grammar

We use a general context-free grammar for the specification of syntactically correct and relevant scenarios. The chosen description format is similar to the extended Backus-Naur-Form [13] and allows alternative, chained, recursive, and repetitive symbol streams. For example, the following fictional grammar describes the set of scenarios which are built from four major components: a prolog, an optimization sequence, a technology mapping sequence, and an epilog.

```

scenario:      prolog optimizing mapping epilog ;
prolog:       "read_file;\n" ;
optimizing:   "DO\n" o_steps "WHILE(" condition "); \n" ;
condition:    "threshold <= 0.9999" ;
o_steps:      o_step o_step
              | o_step ;
o_step:       " remove_redundant;\n"                (7)
              | " kernel_factor;\n"                (2)
              | " cube_factor;\n"                  (3)
              | " transduction;\n"                 (5)
              | " shake_network(" method ");\n"      (7) ;
method:       "forward"
              | "backward" ;
mapping:      [m_step] 1:3 "map(register);\n"
              "map(remainder);\n" ;
m_step:       "map(oai);\n"
              | "map(aoi);\n"
              | "map(mux);\n" ;
epilog:       "write_result;\n"                    (99)
              | "dump_core;\n"                    (1) ;

```

A recursive notation specifies a variable number of optimization steps within the DO ... WHILE construct. For the mapping section, sequences from one to three mapping steps can be used, followed by two standard steps. The integer values in parenthesis are used to control the probabilities of alternative branches. For examples, let's assume 100 scenarios are randomly generated from the grammar "scenario". Statistically, 99 scenarios will end with the transform "write_result" and one with "dump_core". The following scenario is an example for a syntactically correct sentence with respect to the grammar "scenario".

```

read_file;
DO
  remove_redundant;
  shake_network(forward);
  transduction;
  remove_redundant;
WHILE(threshold <= 0.9999);
map(register);
map(remainder);
write_result;

```

The grammar can be translated into a directed graph $G(N, E, r)$ where N is the set of nodes corresponding to terminal and nonterminal symbols of the grammar, $E \subseteq (N \times N)$ the set of directed edges reflecting the productions, and $r \in N$ the unique root of the grammar.

Figure 1a shows an example of a grammar and the corresponding graph G . Each node $n \in N$ is assigned: (1) a token $t(n)$ (empty for nonterminals), (2) a lower bound $l(n)$ and an upper bound $u(n)$ for its repetition, and (3) a node attribute $a(n) \in \{OR, AND\}$ which corresponds to the alternative and chained usage of grammar terminals/nonterminals respectively.

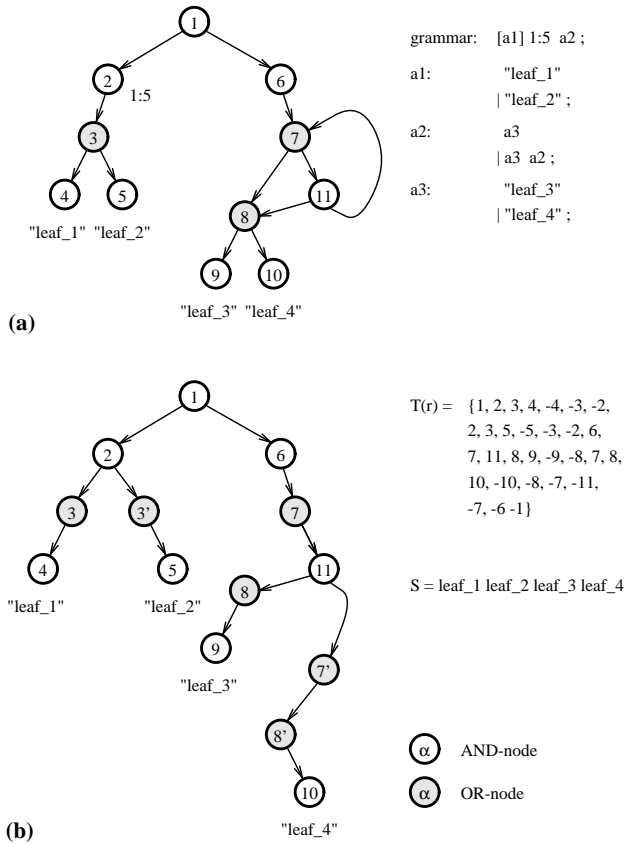


Figure 1: Example for grammar: (a) grammar graph G , (b) parse tree with corresponding grammar trace and sentence.

4 Representation of Scenarios

Depending on the structure of G and the tokens $t(n)$, a syntactically valid sentence corresponds to one or (if the grammar is ambiguous) more parse trees of that grammar [13]. The parse tree can be regarded as a trace through G according to the choices made at the OR-nodes and the number of repetitions picked from the repetition range $[l(n), u(n)]$ at each node. Figure 1b shows an example of a parse tree for the grammar from figure 1a and the corresponding sentence S .

Let $S(n) = \{n' \mid (n, n') \in E\}$ denote the set of successor nodes of n and $\{\alpha_n, n \in N\}$ be a unique positive integer numbering for all nodes. We define a grammar trace $T(n)$ of node n with k successor nodes $S(n) = \{n_1, \dots, n_k\}$ as follows:

$$\begin{aligned}
T(n) &= (T_i(n)), \quad i \in [l(n), u(n)]; \\
T_i(n) &= (T_1(n), T_{i-1}(n)) \quad \text{for } i > 1; \\
T_1(n) &= \begin{cases} (+\alpha_n, -\alpha_n) \\ \text{for } S(n) = \emptyset; \\ (+\alpha_n, T(n_j), -\alpha_n), \quad j \in [1, k] \\ \text{for } a(n) = OR \wedge S(n) \neq \emptyset; \\ (+\alpha_n, T(n_1), T(n_2), \dots, T(n_k), -\alpha_n) \\ \text{for } a(n) = AND \wedge S(n) \neq \emptyset; \end{cases}
\end{aligned}$$

A grammar trace from the root r corresponds uniquely to a parse tree. The trace can be generated by tracking the tree in a depth first traversal recording $+\alpha_n$ when climbing down and $-\alpha_n$ when climbing up. Figure 1b shows the grammar trace for the given example. Each syntactically valid sentence represented by a grammar corresponds to at least one parse tree. Therefore, the set of grammar traces starting from the root covers all possible sentences. The sentence for a given trace can be easily derived by traversing it from left to right and printing the token $t(n)$ for each positive entry α_n .

5 Genetic Operators

The results of genetic operations must be syntactically correct, which imposes certain restrictions on these operations. By defining them based on the parse tree the syntactical correctness of resulting sentences is guaranteed. The idea is, that: (1) Parse subtrees correspond to successive tokens in the sentence and are therefore suitable for local changes. (2) A subtree represents a syntactically self-contained part of the sentence with respect to a node in the grammar graph G . It can consistently be replaced by alternative subtrees belonging to the same grammar node.

Instead of working directly with parse trees, we use grammar traces as an internal representation for sce-

narios. We will introduce subtraces as parts of grammar traces. A subtrace is the counterpart of a parse subtree. It is used to identify trace parts which can consistently be manipulated as described in the following.

Let $T(r)^i$ denote the i th element of the grammar trace $T(r)$. A subtrace $T(r)^{x,y}$ is defined as the subset $(T(r)^x, \dots, T(r)^y)$ of $T(r)$. $T(r)^{x,y}$ is called *balanced* if and only if:

$$T(r)^x = -T(r)^y, \quad (1)$$

$$\sum_{i=x}^y T(r)^i = 0, \quad \text{and} \quad (2)$$

$$\sum_{i=x}^j T(r)^i \geq 0, \quad x \leq j \leq y. \quad (3)$$

For the given example in figure 1b the subtrace $T(r)^{21,26} = \{7, 8, 10, -10, -8, -7\}$ is balanced whereas $T(r)^{15,26} = \{7, 11, 8, 9, -9, -8, 7, 8, 10, -10, -8, -7\}$ is not. We will use the following theorem for the mutation and crossover operator (proof is provided in [14]).

Theorem: Let $T(r)$ be the grammar trace of some sentence of G and $T(r)^{x,y}$ be a subtrace of $T(r)$. If $T(r)^{x,y}$ is balanced, then there exists a grammar trace $T(n)$ for a node n such that $T(n) = T(r)^{x,y}$.

In other words, balanced subtraces can be used to identify parts of traces which were originated by a given grammar node. Also, each balanced subtrace belongs to a unique parse subtree and is directly associated with a part of the corresponding sentence. For example, $T(r)^{21,26}$ corresponds to the subtree starting at node $7'$ of the parse tree in figure 1b and is associated with the sentence part “leaf_4”.

The following sections introduce three basic operations, namely *random_trace*, *mutate_trace*, and *crossover_trace* corresponding to the genetic operators.

5.1 Random Initialization

The random creation of initial scenarios is based on the definition of grammar traces. The following recursive algorithm starts from the root r of the grammar graph G and generates a random trace by depth first traversal, picking random children for OR-nodes and random numbers for variable ranges of node repetitions.

```

random_trace ( $n$ ) /* let  $S(n) = \{n_1, \dots, n_k\}$  be the set
of  $k$  successor nodes of  $n$  */
Begin
   $count = \text{rand\_int}(l(n), u(n));$ 
  For  $i = 1$  To  $count$  Do
     $\text{print}(+\alpha_n);$ 
    Case  $a(n)$  Of
      OR:  $which = \text{rand\_int}(1, k);$ 

```

```

      random_trace( $n_{which}$ );
    AND: For  $j = 1$  To  $k$  Do
      random_trace( $n_j$ );
    End For;
  End Case;
   $\text{print}(-\alpha_n);$ 
End For;
End;

```

For handling probabilities at OR branches of the grammar (see example for grammar “scenario”), the algorithm has to be modified slightly. Instead of using an equal distribution, probabilities according the specified values are chosen (roulette wheel selection [10]). This mechanism can be used to speed up the genetic search by emphasizing certain parts of the grammar which are expected to perform well.

5.2 Mutation

The mutation operator takes one individual and changes a randomly chosen piece of it. The idea is to keep most of the properties of an individual but manipulate some other which might improve its fitness. The mutation operator selects a randomly balanced subtrace of a grammar trace and replaces it by a newly generated trace from the corresponding node.

```

mutate_trace ( $T(r)$ ) /*  $l$  is the length of  $T(r)$  */
Begin
   $L = \text{list all balanced } T(r)^{x,y} \text{ of } T(r);$ 
  pick a random  $T(r)^{x,y}$  from  $L$ ;
   $T'(r)^{1,x-1} = T(r)^{1,x-1};$  /* copy head */
   $T'(r)^{x,y'} = \text{random\_trace}(T(r)^x);$  /* random part */
   $T'(r)^{y'+1,y'+l-y} = T(r)^{y+1,l};$  /* copy tail */
  return  $T'(r)$ ;
End;

```

To avoid mutation operations which do not change the scenario, the selection of balanced $T(r)^{x,y}$ should only include traces belonging to OR nodes or nodes which have variable repetition ranges. All other nodes have no alternative trace and would therefore not be affected by mutation. Furthermore, if variable ranges are used, the generation of a new random subtrace has to ensure that the specified lower and upper bounds for repetitions will not be exceeded.

Figure 2a shows a mutated parse tree derived from the tree of figure 1b. It also gives the corresponding grammar trace and sentence. The subtrace $T(r)^{21,26} = \{7, 8, 10, -10, -8, -7\}$ was selected for mutation and replaced by $T(r)^{21,34} = \{7, 11, 8, 10, -10, -8, 7, 8, 9, -9, -8, -7, -11, -7\}$.

5.3 Crossover

The idea of the crossover operation is to exchange some properties of two individuals resulting in two children. This might cause the combination of advantageous features and lead to a better adaptation of one

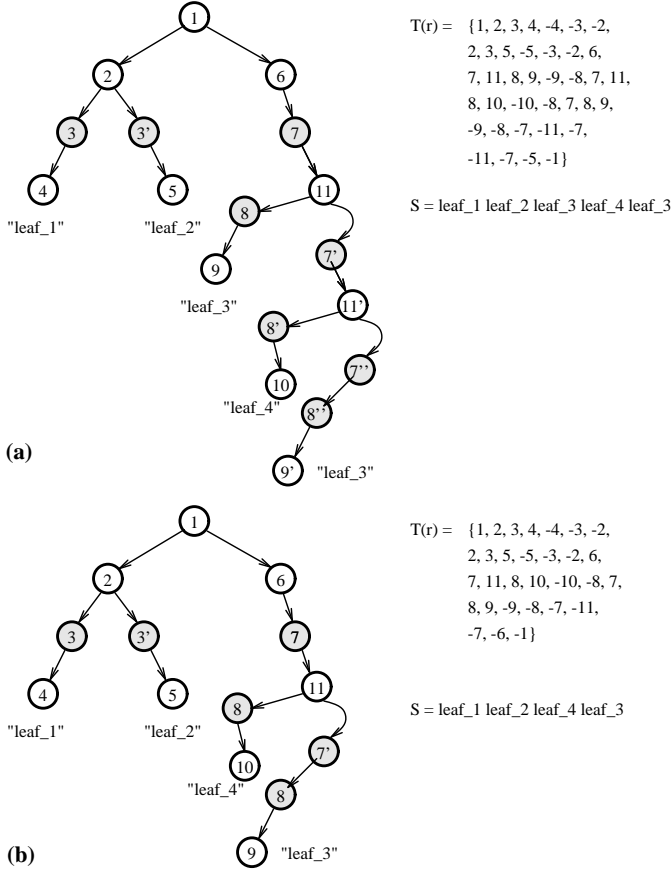


Figure 2: Result of genetic operations: (a) mutation of trace from figure 1b, (b) crossover between traces from figure 1b and figure 2a.

offspring. In our case two randomly chosen subtraces of the parents which belong to the same original grammar node are swapped. The usage of variable repetition ranges excludes pairs which would lead to a violation of the given boundaries.

```

crossover_trace ( $T(r)$ ,  $T'(r)$ )
  /*  $l$  and  $l'$  are the length of  $T(r)$  and  $T'(r)$  */
  Begin
     $L$  = list all balanced  $T(r)^{x,y}$  of  $T(r)$ ;
     $L'$  = list all balanced  $T'(r)^{x,y}$  of  $T'(r)$ ;
    pick randomly proper pair  $(T(r)^{x,y}, T'(r)^{x,y})$ 
      with  $T'(r)^x = T(r)^x$ ;
    exchange  $T(r)^{x,y}$  and  $T'(r)^{x,y}$  in  $T(r)$  and  $T'(r)$ ;
    return  $T(r), T'(r)$ ;
  End;

```

Figure 2b shows one offspring of the crossover operation between the sentences of figure 1b and figure 2a. The subtraces $T(r)^{15,28} = \{1, 11, 8, 9, -9, -8, 7, 8, 10, -10, -8, -7, -11, -7\}$ from figure 1b and $T(r)^{21,34} = \{7, 11, 8, 10, -10, -8, 7, 8, 9, -9, -8, -7, -11, -7\}$ from

figure 2a were swapped. The sentence of the other offspring is: leaf_1 leaf_2 leaf_3 leaf_3 leaf_4.

In order to favor crossover between longer subtraces, we weigh the selection of subtraces by their length.

6 Application

We implemented a genetic algorithm for the optimization of general scenarios based on the presented grammar approach. The implementation is based on a network of IBM RS/6000 workstations communicating via a shared file system. The genetic algorithm runs as a master on a central server while the evaluation of individual scenarios is done in parallel on several workstations.

The concept is applied to optimize synthesis scenarios of the BooleDozer [15] synthesis system. Standard scenarios which were created manually or generated by some previous optimization run can be used as initial search points. To develop a good general scenario, the evaluation of each individual is done for a number of benchmarks. For evaluation purposes the performance of each benchmark is normalized to the standard scenario which was used before. Depending on the optimization goal, the area, delay, and run time criteria are weighed differently.

The development of a general scenario consumes a great deal of CPU time. For example, 10 benchmark circuits, 25 individuals per generation, and 50 generations require 12500 synthesis jobs to be performed. However, their evaluation can be done in parallel on up to 250 workstations in this case. Furthermore, once a good set of scenarios is generated, it can be used as a starting point for further improvements, updates, or for creating a scenario which is specialized to certain circuit classes. This takes far less CPU time and can practically be done overnight. Another way to reduce the computational expense is to first optimize a set of scenarios for the individual benchmarks and only then use the results as initial scenarios for a global optimization run.

As we will show in the result section, there is a great potential for optimizing scenarios to specific circuits or classes of circuits. Although their application scope is limited, specially adapted scenarios can significantly outperform standard scenarios. For crucial designs, specialized scenarios might be the only way to achieve desired results in terms of area or delay. Circuit designs are usually done incrementally over a certain period of time. The synthesis scenario can be optimized gradually along this process so that it is constantly adapted to the current design point. The resulting cost is small since one or only few circuits

are used for the evaluation and the circuit characteristics do not change rapidly between successive scenario optimizations.

7 Results

In a first experiment, a general synthesis scenario for technology-independent optimization was developed with respect to the average performance for 13 benchmarks [16]. The results were compared with the standard scenario. This scenario was carefully designed by a synthesis expert who has specific knowledge about the individual transforms. This standard scenario represents the designers current best knowledge of the interactions between the transforms and how to sequence them into produce a powerful synthesis procedure.

Twenty-five workstations were employed for the evaluation of the scenarios. After 115 generations 37373 synthesis jobs were performed consuming a total of 2300 CPU hours (normalized to a 62 MIPS RS/6000). The results for the optimized scenario versus the original standard scenario are given in table 1. The optimized scenario outperforms the standard scenario by an average of 8 % in the resulting size for which 44 % more CPU time has to be invested.

To test the limitations of the approach, we performed this experiment in a brute force manner. The scenario was generated from scratch, the genetic search was not initialized by some standard scenario. Also, we used some relatively large benchmarks during the entire optimization process which might not be necessary.

As already mentioned in the previous section, it might be far more efficient to first optimize a set of scenarios for individual circuits and then merge them in a final optimization run where all benchmarks are used for evaluation. Smaller tests confirm this proposition. However, no further exploration of this proposition was performed.

A second experiment explored how far synthesis scenarios can be optimized for a particular design. The genetic algorithm was performed with only one benchmark circuit for the evaluation and was initialized with the scenario obtained from the first experiment. Figure 3 shows the performance of the resulting scenarios normalized to the generally optimized scenario. In addition to the target circuit, the performance for the other circuits is shown.

Clearly, synthesis scenarios can be effectively tuned to specific designs or classes of designs. For example, a specialized scenario for the “C432” benchmark outperforms the standard scenario by 31 % and the generally optimized scenario by 13 %. It is interesting to note

Circuit	Standard scen.		Optimized scen.		Comparison (%)	
	# Pins	CPU (sec)	# Pins	CPU (sec)	Δ Pins	Δ CPU
C2670	3025	343	2583	536	-14.6	56.6
C432	743	60	590	100	-20.6	66.8
C880	1247	105	1194	122	-4.3	17.1
alu4	2089	250	1814	663	-13.2	165.6
apex6	2450	201	2350	208	-4.1	3.3
apex7	861	47	789	54	-8.4	14.4
des	9851	1395	9054	4573	-8.1	227.9
frg1	419	60	401	64	-4.3	7.0
ldd	280	23	238	16	-15.0	-28.6
pcl	249	13	234	13	-6.0	-3.6
rot	2211	187	2261	241	2.3	28.8
set	294	27	296	20	0.7	-25.3
term1	1128	161	1027	234	-9.0	45.8
Average					-8.0	44.3

Table 1: Results for the set of benchmarks used for optimizing the synthesis scenario (CPU time is normalized to a 62 Mips RS/6000).

that the creation of specialized scenarios starting from the generally optimized scenario takes far less effort. For example, the tuning of the C432 scenario needed only 9 generations (225 synthesis jobs).

8 Conclusion

We presented an algorithm for the optimization of synthesis scenarios based on a genetic algorithm.

The set of relevant scenarios is described by a general context-free grammar. For such grammars we developed effective genetic operators for the implementation of an evolutionary search scheme. The application of this grammar-based approach is not limited to the optimization of synthesis scenarios. We expect that it will have wider applicability in other fields where optimization problems can be expressed on a formal language basis.

We showed that the proposed approach can significantly optimize manually designed synthesis scenarios. The tuning of generally applicable scenarios requires massive computing resources, although some improvements may be possible. The scenario evaluation during the genetic search can be done in parallel on a large number of machines, which leads to an reduction of the elapsed time of the experiment. We further proved that the quality of synthesis scenarios depends highly on the circuit structure to which they are applied. By tuning scenarios to particular designs, remarkable improvements can be obtained.

A further decrease of the CPU costs is expected by: (1) a stepwise scenario optimization starting with one or only few benchmarks for the evaluation and gradually merging the resulting scenarios in a global optimization run, (2) reducing the number of benchmarks

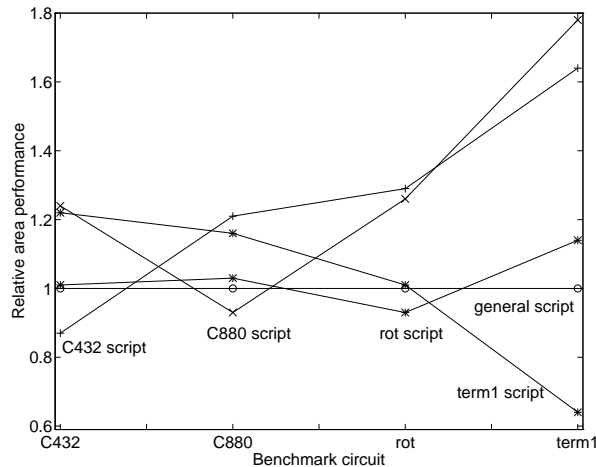


Figure 3: Comparison of the area performance of specializes scenarios versus the generally optimized scenario.

to those that are typical for synthesized designs, and (3) decreasing the size of the benchmarks for evaluation. It is not clear if large benchmarks are necessary. Small test circuits with certain structures might be sufficient.

References

- [1] J. A. Darringer, D. Brand, J. V. Gerbi, W. H. Joyner, and L. H. Trevillyan, "Logic synthesis through local transformations," *IBM Journal on Research and Development*, vol. 25, pp. 272–280, July 1981.
- [2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 1062–1081, November 1987.
- [3] S. Kundu, L. M. Huisman, I. Nair, V. Ivengar, and L. Reddy, "A small test generator for large designs," in *Proceedings of the International Test Conference*, pp. 30–40, 1992.
- [4] I. Rajski and J. Vasudevamurthy, "Testability preserving transformations in multi-level logic synthesis," in *Proceedings of the International Test Conference*, pp. 265–273, 1990.
- [5] K.-C. Chen, "Efficient sum-to-one subsets algorithms for logic optimization," in *Proceedings of the 29th Design Automation Conference*, pp. 443–448, June 1992.
- [6] M. Pedram and N. Bhat, "Layout driven technology mapping," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 99–105, June 1991.
- [7] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan, "Efficient techniques for timing correction," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 415–419, 1990.
- [8] H. Savoj, H.-Y. Wang, and R. K. Brayton, "Improved scripts in MIS-II for logic minimization of combinatorial circuits," in *Proceedings of the M.C.N.C. Workshop on Logic Synthesis*, April 1991.
- [9] J. Holland, *Adaption in Natural and Artificial Systems*. Ann Arbor: Univ. of Michigan Press, 1975.
- [10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading: Addison-Wesley, 1989.
- [11] M. Pipponzi, "MDRIVER: A strategy generation for multiple-level optimization," in *Proceedings of the M.C.N.C. Workshop on Logic Synthesis*, April 1991.
- [12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, no. 220, pp. 671–680, 1983.
- [13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley Publishing Company, 1986.
- [14] A. Kuehlmann and L. P. P. van Ginneken, "Grammar-based optimization of synthesis scenarios," Tech. Rep. Computer Science, RC 18838 (#82376), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, March 1993.
- [15] D. S. Kung, R. F. Damiano, T. A. Nix, and D. J. Geiger, "BDDMAP: a technology mapper based on a new covering algorithm," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, (Anaheim, CA), pp. 484–487, June 1992.
- [16] S. Yang, "Logic synthesis and optimization benchmarks user guide, version 3.0," tech. rep., MCNC Technical Report, January 1991.