On Using Permutation of Variables to Improve the Iterative Power of Resynthesis

Petr Fiser, Jan Schmidt
Faculty of Information Technology, Czech Technical University in Prague
fiserp@fit.cvut.cz, schmidt@fit.cvut.cz

Abstract

Recently we have observed, that behavior of many contemporary logic synthesis and optimization processes depends on variable ordering in their input; they produce different results for different variable orderings. This fact can be exploited to escape local optima in the iterative resynthesis process, where individual synthesis and optimization steps are run repeatedly, in order to gradually improve the solution quality.

In this paper we show an experimental analysis of influence of variable ordering on the result quality, for different synthesis steps in ABC. Next, we present a method of using random permutations of variables in the overall iterative synthesis process, in order to improve the result quality. Experimental evaluation using both standard benchmarks and industrial circuits is presented, to show the viability of the concept.

1 Introduction

Basic principles of logic synthesis of Boolean networks have been established already in 1960's. The synthesis consists of two subsequent steps: the technology independent optimization and technology mapping.

The technology independent optimization starts from the initial circuit description (sum-of-products, truth table, multi-level network) and tries to generate a minimum multilevel circuit description, such as a factored form [5], And Inverter Graph (AIG) [6], [7] or a network of BDDs [8], [9]. Then the technology mapping follows [10]-[13].

The synthesis process, where the forms of its input and output are the same (Boolean networks, AIGs), is called *resynthesis* [14]. Thus, by resynthesis we understand a process modifying the circuit in some way, while keeping the format of its description.

The academic state-of-the-art logic synthesis tool is ABC [15] from Berkeley, a successor of SIS [16] and MVSIS [17]. Individual resynthesis processes in SIS and ABC are represented by commands. Since the number of available resynthesis processes is large (e.g. don't care based node simplification [18], rewriting, refactoring, resubstitution [7], [14], [19], etc.), it is difficult to determine a universal sequence of these commands leading to optimum results. Thus, different *synthesis scripts* were proposed (e.g. "`script.rugged`" and "`script.algebraic`" in SIS, "`resyn`" scripts, "`choice`", and "`dch`" in ABC). These scripts are supposed to produce satisfactory results.

The resynthesis process may be iterated, to further improve the results. Iteration of resynthesis was proposed in ABC [15], too. Authors of ABC suggest repeating the sequence of the technology independent optimization (e.g. the "`choice`" script) followed by technology mapping several times. Also the synthesis process of SIS may be efficiently iterated. The necessary condition for using iteration is that the network structure must not be completely destroyed in the process, e.g., by collapsing it into a two-level form or turning it into a global BDD [8], [9]. Then all the effort made in previous iteration would be in vain. Fortunately this is not the case of the mentioned synthesis scripts.

Even though iteration is not too positively accepted by industry for longer runtimes imposed, it can be advantageously exploited in specific designs, like the low-power or low-area ones. Next, iteration may show the complexity *upper bounds*. By this, efficiency of any synthesis processes can be judged.

In a typical iterative resynthesis, the result quality (size, delay) gradually improves in time, until it converges to a stable solution. In an ideal case it reaches the global optimum. However, the process usually quickly converges to a local optimum, which is sometimes far from the global one (see

Subsection 4.3). Thus, introducing some kind of *diversification*, as known in other iterative optimization processes [20], [21], could be beneficial.

Most of synthesis processes in ABC are greedy and not systematic. Thus, they use some heuristic function to guide the search for the solution. Even though the heuristic is usually deterministic, there often are more equally valued choices. In such situations, the first occurrence is taken. Note that these choices are equally valued just at the point of decision and they will most likely influence the subsequent decisions. Therefore, they can produce different results.

We have realized that many of these processes are also not immune to variable ordering of the source function (source file). Therefore, different runs of one process with different variable ordering produce different results. We take an advantage of this, in order to diversify the search for the solution.

A method of using *random permutations* of input and output variables is proposed in this paper. The order of variables is randomly changed at the beginning of each iteration. Thus, *randomness* is painlessly introduced into the process.

We have run extensive experiments both on standard academic benchmark circuits [22], [23] and industrial designs from OpenCores [24]. We have reached positive average improvements, both in area and delay, for any number of iterations the synthesis was run for.

A similar approach, where randomness was introduced "from outside", was published in [25] and [26]. Here randomly extracted *large parts* of the circuit are synthesized separately, in an iterative way, too. We must admit that the method presented in this paper is inferior to [26], in terms of the result quality. This is obvious, since the method based on variable permutations is theoretically a subset of [26]. However, extraction of the parts involves some computational overhead. Since the random permutations are made in time linear with the number of variables, no noticeable time overhead is involved. Therefore, the main message of this paper is to document that using random permutations always pays off.

2 Discussion on Variable Ordering

Many logic synthesis and optimization processes are sensitive to the ordering of variables in the source function (network) description. Here we discuss possible reasons for it. Experimental results will be presented in Section 4.1.

Typically, variables are processed in a lexicographical order, which is defined a-priori, usually by their order in the source file. Then, different variables orders may induce *heuristic* algorithms run differently, possibly producing different (but definitely correct) results.

A typical and well known example of such a behavior are BDDs [8], [9]. Here the variable ordering is essential; the BDD size may explode exponentially with a "bad" variable ordering [9]. Computing the optimum variable ordering is NP-hard itself, thus infeasible in practice. Even though there are efficient heuristics for determining a possibly good variable ordering [27], they consume some time, whereas do not guarantee any success, and thus they are usually not employed in practice. Typically, the default variable ordering in the BDD manipulation package CUDD [28] (which is used in SIS and ABC, too) is just equal to the variable ordering in the source file.

Most of ABC algorithms are based on processing AIGs [6], [7]. Usually, the AIGs are traversed deterministically, in topological order [7], [19]. But still, there remains some freedom in choosing the order which will be the nodes processed in, since there usually are more nodes in each topological level. In ABC, nodes with the lowest ID (which is determined by the node creation instant) are processed first. Even the nodes creation order may influence the size and topology of the resulting AIG, which affects all the subsequent processes.

Also the well-known two-level Boolean minimizer Espresso [29] (which is used both in SIS and ABC) is sensitive to variable ordering. There are many essential parts of the overall algorithm, where decisions are made in a lexicographical way. Some decisions do not influence the result quality; they just may influence the runtime (e.g., in the tautology checking process [29]), some do influence the result as well (e.g., the Irredundant phase [29]).

Keeping this in mind, all these algorithms that claim to be *deterministic* are not deterministic at all, actually. The initial variable ordering shall be considered as random as any other random ordering. But anyway, the algorithms should be designed to succeed under any ordering. Therefore, introducing random ordering to the synthesis process should not make the process perform worse. Conversely, it could help us escape local minima. From the search space point of view, the global optimum is approached from different sides.

3 The Proposed Method

The state-of-the-art iterative process, as used in ABC [15], can be described as follows: first, the internal description (SOP, AIG, Boolean network, network of BDDs, etc.) for the technology independent optimization is generated from the initial description or the mapped network. Then the technology independent optimization, followed by technology mapping is performed. The process is repeated (iterated), until a stopping condition (number of iterations, result quality, timeout) is satisfied (see Figure 1:).

```
do {
   generate_internal_representation
   technology_independent_optimization
   technology_mapping
} while (!stop)
```

Figure 1: The iterative resynthesis

Assuming that each iteration does not deteriorate the solution, the solution quality improves in time. This needs not be true in practice, however. For such cases several options are possible:

1) to hope that the overall process will "recover" from small deteriorations,
2) to accept only improving (non-deteriorating) changes,
3) to record the best solution ever obtained and return it as the final result,
4) combination of 1) and 3).

The first and the last option are usually used in practice.

For the purpose of this paper, we offer just a slight modification of the algorithm from Figure 1:

```
do {
   randomly_permute_variables
   generate_internal_representation
   technology_independent_optimization
   technology_mapping
} while (!stop)
```

Figure 2: The iterative resynthesis with random permutations

Here we only added the `randomly_permute_variables` step, where the ordering of variables (inputs, outputs, or both) is performed. This step can be executed in a time linear with the number of variables, hence it does not bring any significant time overhead.

4 Experimental Results

4.1 Influence of Permutation on Synthesis Commands

Here we will present an experimental evaluation of some basic synthesis and technology mapping commands in ABC [15], technology independent optimization scripts (which are usually using the basic synthesis commands), and complete synthesis scripts, targeted to standard cells (the "`strash; dch; map`" script) and LUTs (the "`strash; dch; if; mfs`" script). Finally, results of Espresso [29] and even Espresso-exact are shown. The dependency on both input and output variables ordering is studied.

The ABC experiments were conducted as follows: 228 benchmarks from the IWLS and LGsynth benchmarks sets [22], [23] were processed. Given a benchmark, its inputs and/or outputs were randomly permuted in the source BLIF file [30] (or PLA for Espresso), the synthesis command was executed, and the number of AIG nodes (^{1}), gates (^{2}), LUTs (^{3}) or literals (^{4}), respectively, was measured. This was repeated 1,000-times for each circuit.

In order to compactly represent all the results, the maximum and average percentages of size differences were computed, over all the 228 circuits. The results are shown in Table 1.

We can observe striking quality differences, especially for the complete (compound) synthesis processes.

Even the numbers of literals produced by Espresso-exact differ, since Espresso-exact guarantees minimality of the number of terms only.

Table 1. Influence of permutation of variables – summary results

	Process	Permuted inputs		Permuted outputs		Permuted both	
		max.	*avg.*	*max.*	*avg.*	*max.*	*avg.*
Technology independent optimization: commands	balance[1]	7.69%	1.04%	11.48%	1.60%	12.50%	2.27%
	rewrite[1]	15.38%	0.68%	19.30%	2.41%	19.13%	2.78%
	refactor[1]	12.07%	0.36%	29.73%	2.49%	29.73%	2.79%
	resub[1]	2.50%	0.06%	20.83%	1.70%	20.83%	1.71%
Technology independent optimization: scripts	resyn2[1]	44.53%	4.60%	52.75%	5.58%	52.69%	7.38%
	resyn3[1]	13.56%	1.57%	22.50%	2.74%	22.66%	3.72%
	choice[1]	34.40%	7.17%	38.14%	7.14%	36.17%	10.13%
	dch[1]	60.53%	10.42%	40.39%	9.33%	60.50%	13.50%
Technology mapping	map[2]	17.09%	1.35%	12.28%	1.93%	17.09%	2.84%
	fpga[3]	0.00%	0.00%	5.26%	0.29%	5.26%	0.29%
	if[3]	0.00%	0.00%	2.88%	0.24%	2.88%	0.24%
Complete synthesis	strash; dch; map[2]	74.38%	8.67%	70.47%	10.52%	86.27%	13.40%
	strash; dch; if; mfs[3]	92.14%	11.50%	85.42%	12.60%	92.02%	14.81%
Two-level optimization	Espresso[4]	34.90%	1.51%	11.82%	1.04%	42.95%	2.11%
	Espresso-exact[4]	0.63%	0.02%	6.06%	0.23%	6.06%	0.24%

Next, detailed results for two particular circuits, belonging to the largest ones of the measured set, *apex2* and *cordic* [22] are shown in Tables 2 and 3. For each process, the minimum, maximum, and average values are presented, together with percentage differences between the minima and maxima. More precise results were computed here; they were obtained from 10,000 runs. Espresso is insensitive to variable ordering for these particular circuits, thus the results are not present.

When observing the results of the individual synthesis processes and the overall synthesis, the behavior of the *apex2* case is expectable. Almost all the synthesis processes are sensitive to variable ordering, and the effect accumulates in the progress.

However, *cordic* is quite a striking example. First of all, this is the circuit responsible for the maximum difference of LUTs in the complete synthesis process "`strash; dch; if; mfs`". Solutions ranging from 27 to 687 LUTs were obtained. But, strangely enough, the standalone synthesis processes ("`balance`", "`dch`", "`if`", "`mfs`") are *not significantly sensitive* to variable ordering (the mapping phase is completely immune). In quantitative measures, the effects of individual processes cannot be combined to obtain such differences in the final design. Therefore, we conclude that some *qualitative* flaws occur in the progress. This effect is rather surprising and worth studying more thoroughly. More strange phenomena can be observed from the table, however, their explanation is out of scope of this paper.

4.2 Results of the Proposed Synthesis Process

Very exhaustive experiments were performed in order to justify the benefit of using random permutation of variables in the iterative process. We have processed 490 benchmark circuits altogether, coming from academic IWLS and LGsynth benchmark suites [22], [23], as well as from large industrial designs from OpenCores [24] (up to 100,000 LUTs). The 4-LUTs-mapping process was chosen for the testing purpose. However, we expect the same behavior for any target technology.

The most recent LUT-mapping synthesis script suggested by the authors of ABC was used: "`strash; dch; if; mfs; print_stats -b`" as a reference. Then, the ABC command "`permute`" randomly permuting both inputs and outputs was implemented and employed, yielding the script "`permute; strash; dch; if; mfs; print_stats -b`". Both scripts were executed 20-, 100-, 1000-, and 5000-times for each circuit, while the best result ever reached was recorded and returned as the solution (this is accomplished by the "`print_stats -b`" command). The numbers of resulting 4-LUTs and the delay (in terms of the longest path) were measured.

Results of all the 490 circuits are shown in Figure 3: and Figure 4:, for area (4-LUTs) and delay (levels), respectively. The scatter-graphs visualize the relative improvements w.r.t. the state-of-the-art (i.e., no permutations used). Positive values indicate an improvement, the negative ones deterioration.

The size of the original mapped circuit, in terms of 4-LUTs, is indicated on the x-axis. Two border cases, 20 and 5,000 iterations are shown here only. Results of 100 and 1,000 iterations lay in-between.

We see that a significant improvement may be reached even when the process is run for 20 iterations. However, also more deteriorating cases are observed. When iterated more, the results become more positive, especially for larger circuits. This is quite obvious, since these circuits usually converge slower (see Subsection 4.3).

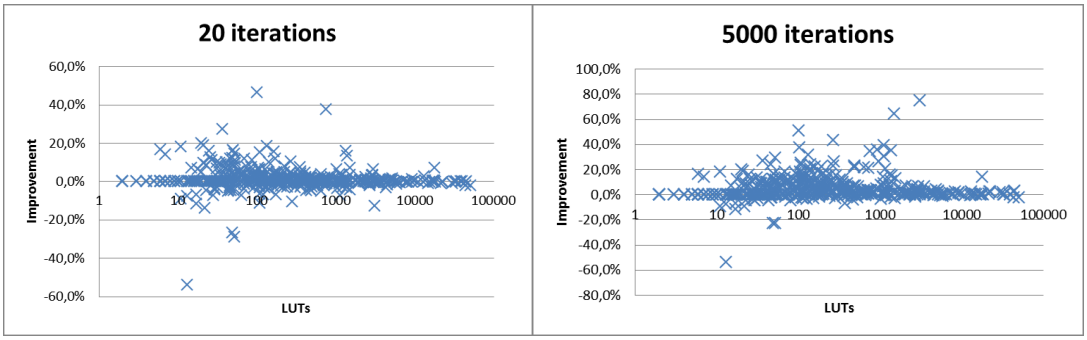

Figure 3: Area improvements w.r.t the standard iterative process

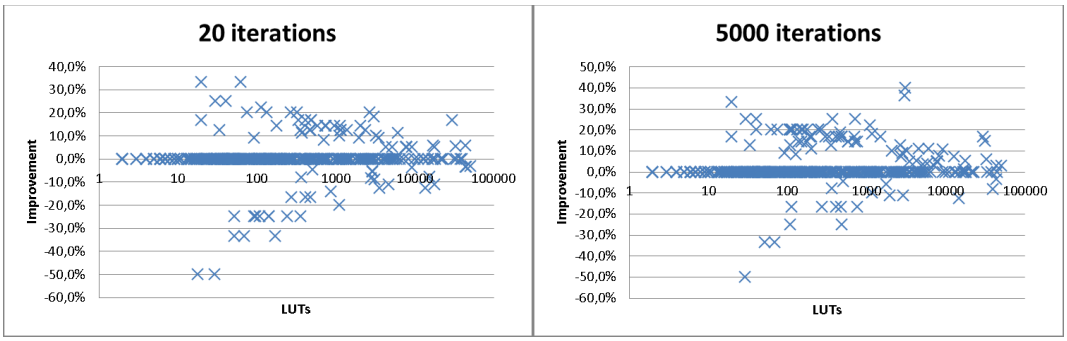

Figure 4: Delay improvements w.r.t the standard iterative process

Summary statistics are shown in Table 4. Only 290 circuits, whose resulting implementation exceeded 100 LUTs, were accounted in these statistics, to make the practical impact more credible. The minimum, maximum and average percentage improvements for both area and delay are given. Also the percentages of cases, where the improvement is positive ("*Better in*") and negative ("*Worse in*") are shown. The complement of the sum of these two values to 100% represents cases where equal solutions were obtained.

Table 4. Summary statistics

Iterations		20	100	1,000	5,000
LUTs	*Minimum*	-12.8%	-8.2%	-5.4%	-6.7%
	Maximum	46.5%	51.2%	74.6%	75.2%
	Average	1.0%	2.1%	4.9%	6.1%
	Better in	52.2%	64.9%	81.0%	82.6%
	Worse in	39.8%	28.8%	15.2%	13.9%
Levels	*Minimum*	-33.3%	-33.3%	-25.0%	-25.0%
	Maximum	22.2%	27.3%	40.0%	40.0%
	Average	0.6%	0.6%	1.6%	2.5%
	Better in	16.3%	13.8%	19.7%	23.9%
	Worse in	9.3%	5.5%	6.2%	5.5%

We see that with an increasing number of iterations results become more stable and tend to improve, both for area and delay. There is a positive average improvement even for 20 iterations. For 5,000 iterations the average improvement reaches 6.1% in area and 2.5% in delay. Also cases, where deterioration was obtained, are becoming rare (13.9% and 5.5% for area and delay, respectively).

Assume the worst case, where the number of deteriorating solutions of one iteration of synthesis is 50% (equal chance for both the improvement and deterioration). In Table 4 we see that all the minimum improvements (maximal deteriorations) are much less than 50%, even for 20 iterations. From these figures we can conclude that permutation always pays off.

4.3 The Convergence Analysis

Here we will show an illustrative example of convergence curves for the iterative synthesis with and without using random permutations for two of the IWLS benchmark circuits [23] *alu4* and *apex2*, see Figure 5:. The progress of the size reduction during 1,000 iterations was traced.

Here we see the justification of our theory. In general, it is not possible to say what method converges faster. Theoretically, both should converge equally fast. This can be seen, e.g., in the *alu4* case, where the standard synthesis converges faster at the beginning, but then the convergence slows down. But more importantly, when the resynthesis without using permutations converges to a local minimum, the permutations will help to escape it (see the *apex2* curves – here the local minimum was reached in the 300th iteration, whereas the solution quality still improves after 1000 iterations when permutations are used). Similar behavior can be observed for most of the tested circuits. This confirms our theory – the *permutations do increase* the iterative power.

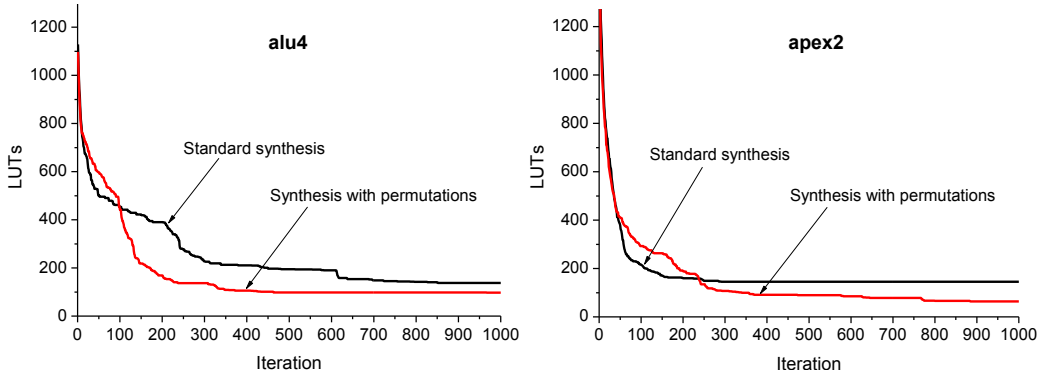

Figure 5: Convergence curves for the *alu4* and *apex2* circuits

5 Conclusions

We have documented that the performance of the state-of-the-art academic logic synthesis tools significantly relies on the variable ordering on their input. Using this fact, we have proposed a method of increasing the iterative power of resynthesis, by non-violently introducing randomness into their run – by randomly permuting input and/or output variables in the process.

The method was tested on standard academic benchmarks and large industrial designs. A positive average improvement in quality (both in area and delay) was obtained. Since introducing the permutations into the iterative process takes almost no time, we can conclude that employing random permutations definitely pays off. Random permutations help to avoid local optima. Cases, where worse results are obtained, are relatively rare.

Now we still have to ask two ultimate questions:

"*What will happen, if I just reorder the variables in the source file header?*" and

"*What shall happen, if I just reorder the variables in the source file header?*"

Acknowledgement

We would like to thank Alan Mishchenko from UC Berkeley for implementing the "`print_stats -b`" option and, more importantly, for a fruitful discussion.

References

[5] G. D. Hachtel and F. Somenzi, "Logic Synthesis and Verification Algorithms", Kluwer Academic Pub. 1996, 564 p.

[6] K. Karplus, "Using if-then-else DAG's for multi-level logic minimization", Univ. California. Santa Cruz, UCSC-CRL-88-29, 1988.

[7] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis”, In Proc. of 43th Design Automation Conference, San Francisco, CA, USA, 2006, pp. 532-535.

[8] S. B. Akers, “Binary decision diagrams”, IEEE Transactions on Computers, vol. C-27, No. 6, June 1978, pp. 509-516.

[9] R. E. Bryant, “Graph based algorithms for Boolean function manipulation“, IEEE Transactions on Computers, vol. 35, No. 8, August 1986, pp. 677-691.

[10] C.W. Moon, B. Lin, H. Savoj, and R.K. Brayton, “Technology Mapping for Sequential Logic Synthesis”, In Proc. of International Workshop on Logic Synthesis, North Carolina, May 1989.

[11] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni Vincentelli, “Logic Synthesis for Programmable Gate Arrays”, In Proc. of the Design Automation Conference, June 1990, pp. 620–625.

[12] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Combinational and sequential mapping with priority cuts”, In Proc. of International Conference on Computer-Aided Design 2007, pp. 354-361.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton, “Improvements to technology mapping for LUT-based FPGAs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26(2), Feb 2007, pp. 240-253.

[14] R. K. Brayton et al., “SAT-based logic optimization and resynthesis”, In Proc. of International Workshop on Logic Synthesis 2007 (IWLS), pp. 358-364.

[15] Berkeley Logic Synthesis and Verification Group, “ABC: A System for Sequential Synthesis and Verification”, http://www.eecs.berkeley.edu/~alanmi/abc/ [Online].

[16] E.M. Sentovich et al., “SIS: A System for Sequential Circuit Synthesis”, Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, Univ. of California, Berkeley, CA 1992.

[17] M. Gao, Jie-Hong Jiang, Y. Jiang, Y. Li, S. Sinha, and R.K. Brayton, “MVSIS”, In the Notes of the International Workshop on Logic Synthesis, Tahoe City, June 2001.

[18] H. Savoj and R.K. Brayton, “The Use of Observability and External Don’t Cares for the Simplification of Multi-Level Networks”, In Proc. of the Design Automation Conference (DAC), 1990, pp. 297–301.

[19] A. Mishchenko and R. Brayton, “Scalable logic synthesis using a simple circuit structure”, In Proc. of International Workshop on Logic Synthesis (IWLS) 2006, pp. 15-22.

[20] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, “Optimization by Simulated Annealing”, Science 13, Vol. 220, no. 4598, May 1983, pp. 671-680.

[21] D. E. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984, p. 41.

[22] K. McElvain, “LGSynth93 Benchmark Set: Version 4.0“, Mentor Graphics, May 1993.

[23] S. Yang, “Logic Synthesis and Optimization Benchmarks User Guide”, Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991.

[24] http://opencores.org

[25] P. Fišer and J. Schmidt, “It Is Better to Run Iterative Resynthesis on Parts of the Circuit”, In Proc. of 19th International Workshop on Logic and Synthesis 2010, Irvine, California, pp. 17-24.

[26] P. Fišer and J. Schmidt, “Improving the Iterative Power of Resynthesis”, In Proc. of 15th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS), 2012, Tallinn (Estonia), pp.-30-33.

[27] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams”, In Proc. of the International Conference on Computer-Aided Design, Santa Clara, CA, 1993, pp. 42-47.

[28] F. Somenzi, \CUDD: CU Decision Diagram Package Release 2.4.1", University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/CUDD [Online].

[29] R. K. Brayton et al., “Logic minimization algorithms for VLSI synthesis”, Boston, MA, Kluwer Academic Publishers, 1984, 192 p.

[30] Berkeley Logic Interchange Format (BLIF), University of California, Berkeley, 2005.

Table 2. Influence of permutation of variables – details for *apex2*

	Process	Permuted inputs				Permuted outputs				Permuted both			
		min.	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*
Technology independent optimization: commands	balance[1]	4162	4191	4174.2	0.69%	4155	4180	4170.6	0.60%	4150	4202	4176.3	1.24%
	rewrite[1]	4129	4137	4132.7	0.19%	4132	4138	4134.8	0.14%	4128	4139	4133.4	0.27%
	refactor[1]	4018	4018	4018.0	0.00%	4018	4027	4022.9	0.22%	4018	4027	4022.8	0.22%
	resub[1]	4302	4317	4309.6	0.35%	4301	4308	4304.4	0.16%	4300	4322	4311.6	0.51%
Technology independent optimization: scripts	resyn2[1]	3360	3448	3399.9	2.55%	3389	3422	3407.4	0.96%	3351	3450	3403.3	2.87%
	resyn3[1]	3918	3945	3927.3	0.68%	3874	3930	3909.8	1.42%	3859	3948	3909.8	2.25%
	choice[1]	4419	4522	4494.0	2.28%	4490	4508	4499.0	0.40%	4419	4524	4492.8	2.32%
	dch[1]	2931	3194	3072.3	8.23%	3008	3143	3067.3	4.30%	2918	3198	3063.5	8.76%
Technology mapping	map[2]	4371	4401	4383.7	0.68%	4354	4383	4371.5	0.66%	4350	4402	4380.1	1.18%
	fpga[3]	2013	2030	2020.1	0.84%	2014	2020	2017.5	0.30%	2006	2029	2016.4	1.13%
	if[3]	2040	2040	2040.0	0.00%	2039	2040	2039.5	0.05%	2039	2040	2039.5	0.05%
Complete synthesis	strash; dch; map[2]	3221	3552	3378.7	9.32%	3292	3464	3360.5	4.97%	3202	3559	3369.8	10.03%
	strash; dch; if; mfs[3]	1502	1731	1631.0	13.23%	1587	1666	1628.3	4.74%	1508	1744	1631.2	13.53%

Table 3. Influence of permutation of variables – details for *cordic*

	Process	Permuted inputs				Permuted outputs				Permuted both			
		min.	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*	*min.*	*max.*	*avg.*	*%*
Technology independent optimization: commands	balance[1]	2727	2735	2730.7	0.29%	2727	2728	2727.5	0.04%	2727	2735	2730.5	0.29%
	rewrite[1]	989	991	990.0	0.20%	987	991	989.0	0.40%	987	991	988.9	0.40%
	refactor[1]	1125	1129	1127.0	0.35%	1128	1128	1128.0	0.00%	1125	1129	1127.0	0.35%
	resub[1]	2723	2723	2723.0	0.00%	2723	2723	2723.0	0.00%	2723	2723	2723.0	0.00%
Technology independent optimization: scripts	resyn2[1]	463	537	502.5	13.78%	487	492	489.5	1.02%	459	541	502.3	15.16%
	resyn3[1]	2677	2724	2695.0	1.73%	2685	2685	2685.0	0.00%	2677	2724	2696.0	1.73%
	choice[1]	2440	2773	2764.0	12.01%	2770	2770	2770.0	0.00%	2440	2774	2761.8	12.04%
	dch[1]	396	545	486.3	27.34%	448	518	482.8	13.51%	411	555	490.7	25.95%
Technology mapping	map[2]	2762	2772	2766.7	0.36%	2765	2766	2765.5	0.04%	2761	2772	2766.2	0.40%
	fpga[3]	930	932	931.0	0.21%	931	931	931.0	0.00%	930	932	931.0	0.21%
	if[3]	804	804	804.0	0.00%	804	804	804.0	0.00%	804	804	804.0	0.00%
Complete synthesis	strash; dch; map[2]	447	2409	567.1	81.44%	486	597	541.2	18.59%	460	2412	571.2	80.93%
	strash; dch; if; mfs[3]	27	687	335.5	96.07%	178	676	425.5	73.67%	34	689	318.4	95.07%