2018 13th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)

Optimum Polymorphic Circuits Synthesis Method

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

Vaclav Simek
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
simekv@fit.vutbr.cz

Abstract— **Polymorphic circuits represent a newly emerging computation paradigm, where one hardware structure is capable to perform two or more different intended functions, depending on instantaneous conditions in the target operating environment. Due to the peculiarity of this paradigm, design of these circuits also calls for a novel approach to logic synthesis procedures. Several attempts to enhance the design of such circuits have already been made, producing highly suboptimal solutions. As an ingenious attempt to set lower bounds on complexity and support designers of sophisticated logic synthesis algorithms, a method with the prospect to facilitate the generation of optimum-size polymorphic circuits is presented in this paper. The core of the proposed method is based on a purposeful exploitation of formal techniques, comprising SAT and PBO in the first place.**

Keywords—polymorphic circuits, Boolean functions, logic synthesis, SAT

I. Introduction

The still ongoing trend of relentlessly scaling diverse circuit features in a close compliance with the Moore's law, while the ubiquitous CMOS technology is approaching its technology limits at the same time, is raising the necessity to introduce rather unconventional technological solutions accompanied by a range of suitable computational paradigms [1]. Thus, it becomes obvious that so called emerging technologies and their properties will play a substantial role in pursuing new generation of devices. Polymorphic circuits can be recognized as one of such examples [2]. Here, by means of using a single hardware structure, two or more intended functions can be implemented. Selection mechanism of the active function at a given moment in time involves the natural occurrence of external conditions, like temperature, supply voltage, light, or even an additional control signal, which have a direct impact on the electrical properties of a particular hardware circuitry.

The motivation behind the activities ultimately resulting in the creation of polymorphic circuits (or similar ones with respect to the general principle of operation) has been originally given by the need to properly address the following aspects: 1) autonomous temperature compensation of a system properties once it is deployed in harsh environments [3] without the possibility of performing regular maintenance, 2) "graceful degradation" of a system [4] that ensures, for example, its safe shut-down or triggers its automatic transition into the low-power emergency operating mode in case of low or depleted batteries. Another application field can be identified in connection with security devices – a hidden function (watermark) can be implemented using polymorphic electronics [4].

The first attempt to design such circuits has been made by NASA Jet Propulsion Laboratory, for adaptively changing the function of a device based on environment temperature [2]-[6]. MOS transistor structures were proposed to accomplish this job and *polymorphic gates* were developed and manufactured, performing distinctive logic functions (e.g., NAND/NOR [5], AND/OR [6]) once exposed to variable temperature ranges.

Due to fundamental constraints projected in the capabilities of existing conventional methods in terms of enabling the design of such structures [4], these polymorphic gates were mostly generated by suitable evolutionary techniques. In particular, the application of genetic programming [5], [6] took place. At a present time, it is possible to design virtually any type of polymorphic gate by using these techniques [7].

Theoretical backgrounds and most importantly the physical availability of polymorphic gates opens a path towards the opportunity to actually design even larger circuits comprising more than just a few logic gates. From a practical point of view, it makes indeed no sense to employ solely the polymorphic logic gates with the sensitivity to the external conditions within a given circuit. It is important to note that a typical circuit with multifunctional or polymorphic behavior is always built around the combination of conventional, mono-functional logic gates, while their polymorphic counterparts are used only for a portion of the circuit structure. There have been already reported numerous attempts to design polymorphic circuits of an arbitrary size [7], [8], [9]. However, all these approaches are considerably suffering from non-optimality of the resulting circuit structure; there has been demonstrated no obvious proof to validate the optimality or a lower bound imposed on size complexity at least.

As a remedy to this, a novel method that facilitates the design tasks of size-optimal polymorphic circuits is presented in this contribution. It is based on application of formal techniques, particularly the Boolean Satisfiability Problem (SAT) [10] and Pseudo-Boolean Optimization (PBO) [11].

The proposed method was tested on standard benchmark circuits, to demonstrate its capabilities.

The paper is structured as follows: Section II presents the related work, Section III contains some preliminaries necessary to understand the problem. Section IV then presents the proposed synthesis method, with experimental results in Section V. Section VI concludes the paper.

II. Related Work

Based on the availability of suitable polymorphic gates, the design of more complex (larger) polymorphic circuits has been tackled recently. One of the approaches is quite straightforward – a polymorphic circuit switching between two functions can be obtained in an easy way by means of implementing these two functions separately, whereas the actual output is selected by a single polymorphic multiplexer [7]. Another approach, based on BDDs [12], was also proposed in [7]. In both situations, the polymorphic gates have been moved to the circuit outputs. This approach eventually leads to very inefficient results, since there is no shared portion of the logic between operating and functional modes of a given circuit.

The first attempt to use sharing of logic resources between the two specified functions has been addressed in [9]. Based on the initial, two-level description of these functions, shared co-kernels [13] are subsequently identified, thus making it possible to move polymorphic gates "deeper" into the circuit structure. However, it is still possible to observe the inclination to place the polymorphic gates near the circuit outputs.

The actual optimality of results obtained by the aforementioned methods is rather questionable at least. Hence, obtaining lower bounds on size complexity of polymorphic circuits, i.e., designing *optimum implementations* of these circuits, is a vital task now.

It is a well-known fact that the Optimum Circuit problem is Σ_2-complete [14]. Needless to say, this observation applies to polymorphic circuits domain as well. The synthesis task supported by the use of formal techniques may be still accomplished with a reasonable efficiency for small circuits.

The problem of obtaining optimum multi-level representations of Boolean functions has been tackled since 1970's. In [15], [16], an approach based on Integer Linear Programming (ILP) has been proposed. Optimum solutions based on NOR gates for functions of up to 4 variables were computed here. Branch and bound techniques were presented in [17], [18], [19], whose scalability is rather doubtful.

A satisfiability (SAT) based approach was introduced in [20], however, it takes aim specifically at the network nodes restricted to majority gates. A SAT and PBO-based method to design optimum XOR-AND-Inverter Graphs has been recently proposed in [21].

One of the most recent works presents the optimum circuit generation for Majority-Inverter Graphs [22], which is based on Satisfiability Modulo Theories (SMT).

Even though the design of optimum circuit implementation is a mature and well-mastered process in case of standard logic, no such approach has been proposed for polymorphic circuits until now. Let us note that because the suggested problem simply exhibits an immensely complex nature, it is possible to design provably optimum implementations only in case of functions with a severely limited number of inputs, typically up to 10. However, this cannot be overcome now by any means (unless a significant breakthrough in circuit complexity theory happens); all the algorithms mentioned above suffer from excessive (double-exponential) growth of asymptotic complexity, either in time or space.

III. Preliminaries

A. Polymorphic Circuit Representation

Multi-output combinational circuits will be considered throughout the paper. A combinational logic circuit can be represented as a directed acyclic graph (DAG), with nodes corresponding to gates (logic functions they implement) and edges representing connections between them. The DAG has one or more roots corresponding to the circuit's primary outputs (POs) and the DAG leaves correspond to its primary inputs (PIs).

Since one of the crucial objectives is to make the optimum circuit generation procedure general enough to be directly applicable to any emerging technology, the set of node functions will not be restricted by any means. In this paper, the concept of nodes will be restricted to 2-input ones only, for the sake of clarity. However, the method can be easily extended to handle nodes with virtually any number of inputs, without the need to introduce any additional principal modifications. Therefore, each node may implement any 2-input function (out of 10 possible).

Similarly to Reduced Boolean Circuits (RBCs) [23], [24] or AND-Inverter-Graphs (AIGs) [25], the edges may be negated, thus indicating the presence of an inverter at the edge.

Besides the role of an ordinary negation, the edges may assume the polymorphic nature. This means, all respective edges are further inverted, when the external *polymorphic stimulus* occurs. The polymorphic stimulus (denoted as *P* in the following text) enables the selection of the circuit operating mode (out of the two intended functions). As a result, there are four types of edges: normal, negated, polymorphic, and negated polymorphic ones.

An example of such a DAG is shown in Fig. 1a, for a polymorphic circuit implementing a function AND/XOR, i.e., the function

$$F = \bar{P}(a \cdot b) + P(a \oplus b),$$

where *P* is the polymorphic stimulus.

In the figure, circle nodes represent an AND gate, hexagon nodes a XOR gate, dashed edges are negated, blue bold edges are polymorphic.

Another example is shown Fig. 1b, for a polymorphic 1-bit full adder. Here, one of the adder inputs is implemented as the polymorphic stimulus. Notice that the polymorphic edges actually represent implicit XOR gates. This is the first hint that efficient implementation of polymorphism may significantly reduce the amount of logic [26].

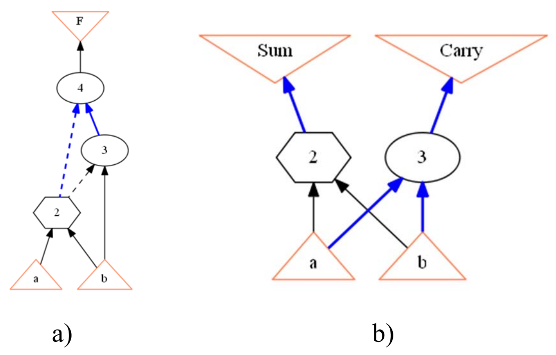

Fig. 1. Examples of polymorphic circuits represented by a DAG, for a) AND/XOR, b) polymorphic 1-bit full adder

B. Boolean Satisfiability (SAT) Problem

The CNF Satisfiability problem (CNF-SAT) [10] is defined as follows: given a Boolean formula in its conjunctive normal form (CNF), find a satisfying assignment of its variables. A *literal* is a variable or its negation. A *clause* is a sum of literals. The CNF is a product of clauses. The formula is satisfiable, when there exists an assignment of its variables, so that the respective functional value is equal to one. In the constructive version of the SAT problem, a satisfiability witness, i.e., the satisfying assignment of variables, is returned as a result.

IV. The Proposed Method

The proposed SAT-based method of designing size-optimal polymorphic circuits is presented here. Any set of 2-input gates can be used as a set of building blocks (DAG nodes), with their costs (area) specified. As stated in Subsection III.A, the DAG edges may be negated and/or polymorphic.

For the purpose of this paper, the procedure generating one size-optimal structure for a given polymorphic function will be presented. However, the procedure can be easily extended to optimize the delay, and/or to enumerate *all* such structures, following the principles given in [21].

A. The Algorithm

The Optimum Circuit Problem is solved by its reduction to a *decision CNF-SAT problem* [10]. Since these problems belong to different complexity classes of polynomial hierarchy (Optimum Circuit Problem is Σ_2-complete [14]), the reduction is not polynomial.

The optimization problem is reduced to its decision version by a simple trick: a decision problem "*Does there exist an n-node implementation of a given k-input function?*" is solved, whereas we start with $n = 1$, and if the answer is negative, n is increased. This procedure is repeated, until a positive answer is obtained. The solution witness is then the *optimum* solution of the original problem [22].

The procedure is outlined by a pseudo-code shown in Fig. 2. The input to the algorithm is a truth table (set of binary vectors, for a multi-output function) of the function to be implemented, the output is its optimal implementation structure.

For a given function *f* having *k* inputs, *o* outputs, and *n* nodes, a characteristic function in CNF is generated (see Subsection IV.B) and SAT is solved [27]. If the SAT is unsatisfiable, *n* is increased by one and the procedure is repeated, until a satisfiable solution is obtained. The satisfiability proof is then the solution: the optimum circuit structure.

```
Generate_structure (truth_table f, int k) {
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      Sol = SAT_Solve(CNF);
      if (Sol.unsat) n++;
   } while (Sol.unsat);
   return Sol.extract_structure;
}
```

Fig. 2. The basic optimum structure generation procedure

B. The CNF Construction

The main procedure of the algorithm, the SAT instance generation (`Generate_CNF` in Fig. 2) is described here.

Let us have a polymorphic circuit with *k* inputs and *o* outputs constructed of n gates. As stated in Subsection III.A, such a circuit can be represented as a DAG with attributed edges. The following variables and constraints are introduced:

- Each primary input (PI) and DAG node has a unique index, $0 \ldots n + k - 1$, where DAG PIs are represented by nodes indexed $0 \ldots k - 1$ and internal nodes are indexed $k \ldots n + k - 1$;
- the DAG has *o* outputs, each can be connected to any node $0 \ldots n + k - 1$;
- the parent node always has higher index than both its children;
- node inputs (exactly two here) are labeled 0 (left input) and 1 (right input);
- each node can implement any function from a given set of 2-input functions **F**. The ordinary numbers of functions are binary-encoded, i.e., *v* variables are needed to index the functions, $v \in \{0 \ldots \lceil \log_2 |\mathbf{F}| \rceil - 1\}$.

For easier understanding of the following explanations, the network with corresponding variables describing the structure (node labels) and function (edge labels), is shown in Fig. 3, for a node *i* (a), connection of a node *j* to the *m*th input of a node *i* (b), and connection of the node *i* to the primary output *w* (c). The circle node represents the node *i*, rectangle nodes describe edge value modifiers (negation, polymorphy), and the diamond-shaped nodes represent the network interconnection.

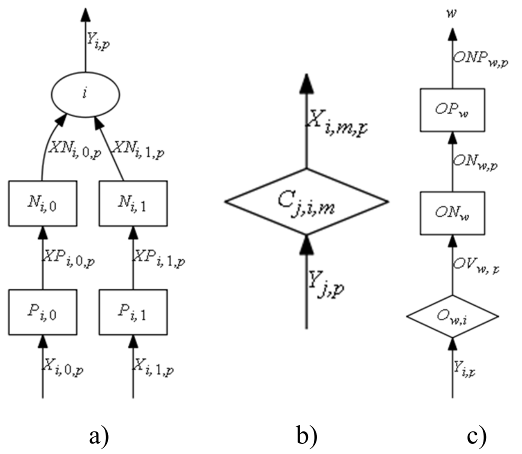

Fig. 3. Network structure with corresponding variables

In the scenario given above, a set of Boolean variables describing the *structure* of the network is defined:

- $N_{i,m}$ ... m^{th} input of node i is negated, defined for $i \in \{k, \ldots, n + k - 1\}, m \in \{0,1\}$;
- $P_{i,m}$ ... m^{th} input of node i is polymorphic, defined for $i \in \{k, \ldots, n + k - 1\}, m \in \{0,1\}$;
- $C_{i,j,m}$ ... output of i^{th} node is connected to m^{th} input of j^{th} node, defined for $i \in \{0, \ldots, n + k - 1\}$, $j \in \{\max(i + 1, k), \ldots, n + k - 1\}, m \in \{0, 1\}$;
- $O_{w,i}$ ... w^{th} output is connected to i^{th} node, $w \in \{0 \ldots o - 1\}, i \in \{0 \ldots n + k - 1\}$;

- ON_w … w^{th}output is connected to a node by negated edge, $w \in \{0 \ldots o-1\}$;
- OP_w … w^{th}output is connected to a node by polymorphic edge, $w \in \{0 \ldots o-1\}$;
- $F_{i,v}$ … the i^{th}node function selector, $i \in \{k, \ldots, n+k-1\}, v \in \{0 \ldots \lceil \log_2 |\mathbf{F}| \rceil - 1\}$.

Next, constraints (SAT clauses) are defined, to describe the network *validity*. Note that the universal quantifiers actually represent conjunctions of SAT clauses. The final CNF-SAT instance is then formed by conjunction of all the constraints.

For the lack of space, the final CNFs will not be presented in some cases; only more intuitive Boolean formulas will be given. These can be easily converted to CNFs using laws of Boolean algebra.

1) Each node input is connected somewhere – to a node with a smaller index (including PIs):

$$\forall i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}: \bigvee_{j \in \{0,\ldots,i-1\}} C_{j,i,m}$$

2) Each node output is connected somewhere – to a node with a higher index or to an output:

$$\forall i \in \{k, \ldots, n+k-1\}: \left(\bigvee_{\substack{j \in \{\max(i+1,k),\ldots,n+k-1\} \\ m \in \{0,1\}}} C_{i,j,m} \right) \vee \bigvee_{w \in \{0 \ldots o-1\}} O_{i,w}$$

3) Each node input has only one source

$$\forall i \in \{k, \ldots, n+k-1\}, j \in \{0, \ldots, i-1\}, h \in \{j+1; i-1\}, m \in \{0,1\}: C_{j,i,m} \Rightarrow \overline{C_{h,i,m}}$$

In simple words, if the m^{th} input of node i is connected to node j, it must not be connected to node h.

4) Each primary output is connected to at least one node

$$\forall w \in \{0 \ldots o-1\}: \bigvee_{i \in \{0,\ldots,n+k-1\}} O_{w,i}$$

5) Each primary output is connected to one node at most

$$\forall w \in \{0 \ldots o-1\}, i \in \{0 \ldots n+k-1\}, j \in \{i+1 \ldots n+k-1\}: O_{w,i} \Rightarrow \overline{O_{w,j}}$$

In simple words, if the output w is connected to node i, it must not be connected to node j.

Next, the desired *function* must be enforced. This means that the network must output the correct functional value for each input combination, i.e., for all 2^k minterms for all outputs.

For this purpose, additional Boolean variables, specific for each $p \in \{0, \ldots, 2^k - 1\}$ minterm, are defined:

- $Y_{i,p}$ … i^{th} node output value, $i \in \{0, \ldots n+k-1\}$. For $i < k$, it represents a primary input (PI);
- P_p … polymorphic stimulus value;
- $X_{i,m,p}$ … m^{th} input of node i; defined for $i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}$;
- $XN_{i,m,p}$ … m^{th} input of node i, after possibly negated edge following the polymorphic edge; defined for $i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}$;
- $XP_{i,m,p}$ … m^{th} input of node i, after possible polymorphic edge; defined for $i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}$;
- $OV_{w,p}$ … w^{th}output node value;
- $ON_{w,p}$ … w^{th}output node value, after possible negation;
- $ONP_{w,p}$ … w^{th}output node value, after possible polymorphic edge, following the negation.

Next, constraints enforcing the function are defined:

6) Polymorphic edges:

$$\forall i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}: XP_{i,m,p} = \left(P_{i,m} \Rightarrow P_p \oplus X_{i,m,p}\right) + \left(\overline{P_{i,m}} \Rightarrow X_{i,m,p}\right)$$

In simple words, if the edge is polymorphic, negate the signal, if the polymorphy stimulus occurs.

7) Negated edges:

$$\forall i \in \{k, \ldots, n+k-1\}, m \in \{0,1\}: XN_{i,m,p} = XP_{i,m,p} \oplus N_{i,m}$$

8) Nodes interconnection:

$$\forall i \in \{0, \ldots, n+k-1\}, j \in \{\max(i+1;k), \ldots, n+k-1\}, m \in \{0,1\}: C_{i,j,m} \Leftrightarrow Y_{i,p} = X_{j,m,p}$$

9) Negated output edges:

$$ON_{w,p} = OV_{w,p} \oplus ON_w$$

10) Polymorphic output edges:

$$ONP_{w,p} = ON_{w,p} \oplus \left(OP_w \wedge P_p\right)$$

11) Outputs:

$$\forall w \in \{0 \ldots o-1), i \in \{0, \ldots, n+k-1\}: O_{w,i} \Rightarrow \left(OV_{w,p} = Y_{i,p}\right)$$

12) Nodes functions:

$$\forall i \in \{k, \ldots, n+k-1\}: Y_{i,p} = XN_{i,0,p} < op > XN_{i,1,p}.$$

In principle, the characteristic functions of the operators $< op >$ (nodes functions) are constructed by deriving their on- and off-sets in CNF. Depending on the function selection (variables $F_{i,v}$), constraints for the outputs ($Y_{i,p}$) and inputs ($XN_{i,0,p}, XN_{i,1,p}$) of nodes are derived, based on these node characteristic functions. Due to the lack of space, we will omit a detailed formal description of the procedure.

Just to give a simple example, if, e.g., $F_{i,0} = 0$ selects an AND node i, these constraints will be generated:

$$\forall i \in \{k, \ldots, n+k-1\}: F_{i,0} = 0 \Rightarrow \left(Y_{i,p} = XN_{i,0,p} \cdot XN_{i,1,p}\right)$$

13) The function – input and output values:

$\forall i \in \{0, \ldots k-1\}: Y_{i,p} =$ forced respective bit value

$P_p =$ polymorphic stimulus value for the p^{th}minterm

$\forall w \in \{0 \ldots o-1)$:

$ONP_{w,p} =$ forced outputs for minterm p

The clauses stated above are concatenated to form a CNF, to produce a SAT instance. A solution of this instance, particularly the values of variables $N_{i,m}, P_{i,m}, C_{i,j,m}, O_{w,i}, ON_w$, OP_w, and $F_{i,v}$, then represent the implementation of the desired DAG.

Note that the number of clauses describing the DAG validity grows linearly with both k and n, but the number of clauses describing the function grows exponentially with k, because of an exponential number of minterms. When combined with an NP-complete SAT-solving repeatedly used in the process, it is clear that this approach is feasible for small k's only. However, it is fully sufficient for some purposes.

C. Employing Optimization

Since any set of 2-input gates can be used to implement node functions, it could be useful to have the means to adjust their cost (area) and use the summary network cost as the optimization criterion, instead of just minimizing the nodes count. This can be easily accomplished by using Pseudo-Boolean Optimization (PBO) [11], [28] instead of SAT. The polymorphic (or inverted) edges may also be given a customizable cost, yielding, e.g., minimization of the number of polymorphic edges in the result. All these features have been implemented in our tool MinCirc [29]. However, detailed description of their principles is beyond the scope of this paper. For some details see [21].

V. Experimental Results

Experimental results are demonstrated in this section. Several smallest circuits from the MCNC [30] and ITC'99 [31] benchmark sets, plus generic adders were used for this purpose. Combinational parts of sequential circuits were extracted. These circuits were collapsed to a PLA by ABC [32] to obtain a truth table. MiniSAT [27] has been used as a SAT-solver.

For the purpose of the experiments, we have defined a scenario, where one circuit input is defined as the polymorphic stimulus, while the other inputs remain the primary inputs. Such a scenario is not that unrealistic – one may imagine a technology, where the function of multiple gates is influenced just by one signal. The Si-NW technology is such an example [33].

Optimum implementations of the example circuits have been synthesized, without polymorphism being used and compared to their respective optimum polymorphic designs, for different inputs set as polymorphic stimuli. The results are shown in TABLE I. for AND-nodes based circuits. Similar results are presented in TABLE II. , with the nodes set extended by a XOR gate (with its cost set equal to the AND gate cost).

TABLE I. Synthesis results – only AND nodes used

			Std. synth.		Polymorphic synthesis		
Name	***k***	***o***	***Nd.***	***Levels***	***Nd.***	***P. edges***	***Levels***
1-adder	3	2	7	4	3	6	2
2-adder	5	3	14	8	10	6	6
c17	5	2	6	3	5	5	3
daio	5	6	10	4	8	9	4
b06	10	14	10	4	11	1	5
lion	4	3	9	5	7	8	4
majority	5	1	8	6	5	6	4
s27	7	4	7	5	7	6	3
t	5	2	6	3	5	5	3
Sum			**288**	**165**	**239 (17%)**	**227**	**145 (12%)**

After the circuit name, numbers of its inputs and outputs (k, o) are given. Then, results of standard optimum synthesis [21] are shown, in terms of the number of nodes (*Nd.*) and levels, followed by the results of polymorphic implementations. The first circuit input was always selected as polymorphic stimulus, the "*P. edges*" column gives the number of polymorphic edges in the implementation.

TABLE II. Synthesis results – AND and XOR nodes used

	Std. synthesis			Polymorphic synthesis			
Name	***Nd.***	***XORs***	***Lev.***	***Nd.***	***XORs***	***P. edges***	***Levels***
1-adder	5	3	3	2	1	6	1
2-adder	10	6	6	7	4	9	4
c17	6	0	4	5	0	5	2
daio	7	3	3	7	2	4	4
b06	8	1	5	9	2	3	5
lion	9	1	7	7	0	8	5
majority	8	0	5	5	0	6	4
s27	7	0	5	7	0	5	5
t	6	0	3	5	0	5	3
Sum	**206**	**25**	**132**	**176 (15%)**	**12 (12%)**	**184**	**112 (15%)**

Results of only 9 circuits are shown in the tables, with summary values over all exercised 31 circuits (with different inputs set as polymorphy stimuli) given in the last rows, together with overall percentage reductions compared to the standard synthesis process.

It is possible to see that the polymorphic implementations exhibit smaller area and delay in most of cases, assuming that the polymorphic edges are "for free". Even though this observation is quite obvious, since the notion of polymorphism actually allows implementing implicit XOR gates (see Section III), the purpose of the experiments was to illustrate *how much* area can be saved by means of using the polymorphic electronics paradigm.

VI. Conclusions and Discussion

A SAT-based method generating optimum polymorphic circuits described by a directed acyclic graph (DAG) was presented. The polymorphism property is implemented by introduction of polymorphic edges into the DAG.

The method is general, in the sense of any set of 2-input gates can be used as the circuit building blocks (DAG nodes). These gates can be assigned an arbitrary cost (size), allowing to produce circuits minimizing this cost. Polymorphic edges may be assigned a cost too.

Since the complexity of the problem solved – the optimum circuit – is immense (it is a Σ_2-complete problem), the method can be applied to functions having typically up to 10 inputs. However, the method can still serve as a mean of obtaining lower bounds on complexity of polymorphic circuits.

The achieved experimental results enable the comparison of "standard" optimum implementations of several benchmark circuits with their polymorphic counterparts. A scenario, where one circuit input serves as a polymorphic stimulus, was used. As a result, polymorphic implementations exhibit an average 17% improvement in the number of gates. However, polymorphism is considered to be available "for free" in this scenario.

Presenting this experimental comparison, however, was not the main purpose of this paper, since it is unrealistic, unless concrete technology is targeted. Instead, the objective was to present the method itself, in its most general form. Then, it can be used, e.g., as a part of more complex synthesis algorithms applied to specific target technologies. For example, it can be used for generating optimum implementations of functions with a limited number of inputs, to be used in general logic optimization processes, like *rewriting* [34], [35]. Here, optimum implementations of 4-input functions are used to replace their functional equivalents in a circuit to be optimized, to reduce its size. Generation of 4-input functions by the proposed method is very fast, indeed. Thus, for such functions it can be applied even on-demand, in the rewriting process. It has been found that using replacement functions of more than 6 inputs is impractical, because of an immense growth of the rewriting algorithm complexity caused by a high number of replacement candidates [34]. Therefore, our algorithm is applicable here.

ACKNOWLEDGEMENT

This work was partially supported by the grant GA16-05179S of the Czech Grant Agency, “Fault Tolerant and Attack-Resistant Architectures Based on Programmable Devices: Research of Interplay and Common Features” (2016-2018). Another support for this work has been gratefully provided by the grant FIT-S-17-3994 of Brno University of Technology, “Advanced parallel and embedded computer systems” (2017-2019). Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme ”Projects of Large Research, Development, and Innovations Infrastructures” (CESNET LM2015042), is greatly appreciated.

REFERENCES

[1] I. L. Markov, "Limits on fundamental limits to computation", Nature, vol. 512, no. 7513, August 2014, pp. 147-154.

[2] A. Stoica, “Polymorphic electronics - A novel type of circuits with multiple functionality,” NASA New Technology Report NPO-21213, 10106/2000.

[3] A. Stoica, “EHW Approach to Temperature Compensation of Electronics,” NASA Tech Briefs NPO-21146, April 2004.

[4] A. Stoica, R.S. Zebulum, and D. Keymeulen, J. Lohn, “On Polymorphic Circuits and Their Design using Evolutionary Algorithms,” in Proc. of 20th IASTED Int. Conf. on Applied Informatics, Insbruck, 2002, 6 p.

[5] A. Stoica, R.S. Zebulum, “Multifunctional logic gate controlled by temperature,” NASA Tech Briefs. Pasadena: California Inst. of Tech., NPO-30795, July 2005, pp. 18.

[6] A. Stoica, R.S. Zebulum, “Polymorphic Electronic Circuits,” NASA Tech Briefs. Pasadena: California Inst. of Tech., NPO-21213, April 2004. p. 10.

[7] Z. Gajda, L. Sekanina, “On Evolutionary Synthesis of Compact Polymorphic Combinational Circuits,” in Journal of Multiple-Valued Logic and Soft Computing, 2011, vol. 17, no. 6, pp. 607-631.

[8] R. Tesar, V. Simek, R. Ruzicka, and A. Crha, ”Design of Polymorphic Operators for Efficient Synthesis of Multifunctional Circuits,” Journal of Computer and Communications, 4, 2016, pp. 151-159.

[9] A. Crha, R. Ruzicka, V. Simek, “Synthesis Methodology of Polymorphic Circuits Using Polymorphic NAND/NOR Gates,” in International Conference on Mathematical/Analytical Modelling and Computer Simulation, Cambridge, UK, 2015, pp. 612-617.

[10] M. R. Garey and D. S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co. New York, USA, 1990, p. 338.

[11] E. Boros and P. L. Hammer, “Pseudo-Boolean optimization,” in Discrete Applied Mathematics, Volume 123, Issues 1–3, 15 November 2002, pp. 155-225.

[12] S. B. Akers, “Binary decision diagrams,”, in IEEE Transactions on Computers, 27(6), June 1978, pp. 509-516.

[13] G. D. Hachtel and F. Somenzi, Logic Synthesis and Verification Algorithms, Boston, MA, Kluwer Academic Publishers, 1996, 564 p.

[14] C. Umans, “The Minimum Equivalent DNF Problem and Shortest Implicants”, Journal of Computer and System Sciences, vol. 63, no. 4, 2001, pp. 597-611.

[15] S. Muroga and T. Ibaraki, “Design of optimal switching networks by integer programming,” IEEE Trans. on Computers, vol. 21, no. 6, 1972, pp. 573–582.

[16] S. Muroga and H. C. Lai, “Minimization of logic networks under a generalized cost function,” IEEE Trans. on Computers, vol. 25, no. 9, 1976, pp. 893–907.

[17] T. Nakagawa, “A branch-and-bound algorithm for optimal AND-OR networks (The algorithm description),” Dep. Comput. Sci., Univ. of Illinois, Urbana, Rep. UIUCDCS-R-71-462, June 1971.

[18] E. A. Ernst, “Optimal combinational multi-level logic synthesis,” PhD thesis, The University of Michigan, 2009.

[19] R. Drechsler and W. Günther, “Exact circuit synthesis,” in Proc. of International Workshop on Logic & Synthesis (IWLS), 1998.

[20] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, “Finding efficient circuits using SAT-solvers,” in Int’l Conf. on Theory and Applications of Satisfiability Testing, 2009, pp. 32–44.

[21] P. Fiser, I. Halecek, and J. Schmidt, “SAT-Based Generation of Optimum Function Implementations with XOR Gates”, in Proc. of 20th Euromicro Conference on Digital Systems Design (DSD), Vienna, Austria, August 31-September 1, 2017, pp. 163-170.

[22] M. Soeken et al. “Exact Synthesis of Majority-Inverter Graphs and Its Applications”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 36, No. 11, 2017, pp. 1842-1855.

[23] P. A. Abdullah, P. Bjesse, and N. Een, “Symbolic reachability analysis based on SAT-solvers,” in Proc. TACAS’00, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2000.

[24] P. Bjesse, A. Borlv, “DAG-aware circuit compression for formal verification,” in IEEE/ACM International Conference on Computer-Aided Design, 2004, pp. 42–49.

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

[26] I. Halecek, P. Fiser, J. Schmidt, “Are XORs in logic synthesis really necessary?,” in Proc. of the IEEE 20th Int. Symposium on Design and Diagnostics of Electronic Circuit & Systems, 2017, pp. 134–139.

[27] N. Een, N. Sorensson, “An extensible SAT-solver,” in Lecture Notes in Computer, Science 2919 - Theory and Applications of Satisfiability Testing. Springer Verlag, 2004, pp. 333-336.

[28] N. Een, N. Sorensson, “Translating Pseudo-Boolean Constraints into SAT”, in Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, pp. 1-26, Nov. 2006.

[29] P. Fiser, “MinCirc: Optimum Circuits Generator”, 2017. Available from: http://ddd.fit.cvut.cz/prj/MinCirc/

[30] S. Yang, “Logic synthesis and optimization benchmarks user guide: Version 3.0,” MCNC Technical Report, Tech. Rep., Jan. 1991.

[31] F. Corno, M. S. Reorda, and G. Squillero, “RT-level ITC’99 benchmarks and first ATPG results,” in IEEE Design Test of Computers, 2000.

[32] A. Mishchenko et al., “ABC: A system for sequential synthesis and verification,” 2012.

[33] L. Amaru et al. “New Logic Synthesis as Nanotechnology Enabler”, Proceedings of the IEEE , vol. 103, no. 11, Nov. 2015, pp. 2168-2195.

[34] K. Brayton, Robert, A. Mishchenko, and S. Chatterjee, “DAG-aware AIG rewriting: a fresh look at combinational logic synthesis,” in 43rd ACM/IEEE Design Automation Conference. ACM, 2006, pp. 532–535.

[35] I. Halecek, P. Fiser, J. Schmidt, “On XAIG Rewriting,” in Proc. of 26th International Workshop on Logic & Synthesis (IWLS), Austin, TX, June 17–18, 2017, pp. 89-96.