SAT-Based Generation of Optimum Function Implementations with XOR Gates

Petr Fišer, Ivo Háleček, and Jan Schmidt
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
fiserp@fit.cvut.cz, schmidt@fit.cvut.cz

***Abstract*— This paper presents a method for generating optimum multi-level implementations of Boolean functions. It is based on Satisfiability (SAT) problem solving, while different SAT techniques are employed to reach different targets. The method is able to generate one, or enumerate all optimum implementations, while any technology constraints can be applied. Results for 4-input functions implemented by XOR-AND-Inverter-Graphs (XAIGs) with different XOR nodes costs are presented. Scalability and feasibility of the method is presented. Finally, an experimental evaluation of XAIG-based rewriting algorithm with optimum replacement circuits is presented and compared with the previous solution.**

Keywords*— Boolean functions, logic synthesis, SAT, PBO, optimum implementations, exact synthesis

I. Introduction

The need for obtaining optimum gate-level implementations of Boolean functions is encountered in many applications, mostly in logic synthesis and optimization. The *rewriting* algorithm [1], where sub-circuits of a network are repetitively replaced by their (preferably) optimum implementations, is the most striking example.

Until now, the replacement circuits for rewriting were produced by using "powerful-enough" synthesis methods [1], [2], which however do not guarantee the real optimum. This process is also time-consuming, and for this reason the optimum implementations of all functions (or their respective NPN classes [3], [4]) of a given number of inputs are pre-computed and stored in memory. Moreover, this approach is not extensible; once new technological primitives are to be considered, or different primitives' costs are to be assumed, all the implementations must be recomputed.

In some cases, having a single optimum implementation of each function is not sufficient. Particularly, even though all optimum implementations (in size and delay) are equal as stand-alone solutions, their incorporation in a network may make difference, mostly because of logic sharing possibilities [1].

Therefore, a more universal way of computing optimum implementations is required. In this paper we propose a method able to use any technological primitives (logic gates) in the implementation, assign arbitrary costs to gates and hereby compute the optimality, generate one or more (or all) solutions, and run on-demand, because of its relatively low run-time.

In its basic principle, the method is based on generating a Satisfiability (SAT) problem instance in a conjunctive normal form (CNF) for a given function, whose solution is the optimum function implementation. Pseudo-Boolean Optimization (PBO) [5] can be then incorporated to accommodate different gates costs. Enumeration SAT-problem solving can be used to obtain more or even all solutions.

The feasibility of the method is evaluated by providing experimental results for up to 12-variable functions. As a case study, two target implementations are assumed: the And-Inverter-Graphs (AIGs [1]), where only AND gates and inverters are used, and XOR-AND-Inverter-Graphs (XAIGs [2]), where XOR gates are assumed as well. Solutions with different costs of XOR gates are produced and their properties evaluated.

The obtained optimum implementations are used as replacement circuits for XAIG-based rewriting [2], [6] and compared with the previously used method based on non-exact, heuristic optimization.

II. Related Work

The problem of obtaining optimum multi-level representations of Boolean functions has been tackled since 1970's. In [7], [8] 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 [10], [11], whose scalability is rather poor.

A satisfiability (SAT) based approach was introduced in [12], however, it is targeted strictly to majority gates as nodes.

The most recent work presents Satisfiability Modulo Theories (SMT) based optimum circuit generation for Majority-Inverter Graphs [13]. This approach is claimed to be scalable enough to be incorporated during rewriting in its run-time.

Basically, all the above-mentioned approaches primarily optimize the gates count and (sometimes) delay. Two exceptions are [8] and [9], where the secondary optimization criterion is the wiring density. In contrast to this, we propose a method where costs of gates are assumed, apart from the gates count and delay.

In this paper we try to stay with a standard SAT-based approach, for which very efficient solvers exist [14], instead of using more time-consuming ILP or SMT solvers.

III. PRELIMINARIES

A. And-Inverter Graphs and XOR-And-Inverter Graphs

And-Inverter Graphs (AIGs) [1], [15], [16] are directed acyclic graphs with one or more roots (representing the outputs), where nodes are two-input AND nodes and edges represent connections between them. Edges may be inverted, meaning that the respective subgraph is negated. This can be understood as an inverter presence on the connection. The AIG outputs may be negated as well.

XOR-And-Inverter Graphs (XAIGs) [2], [6] are a generalization of AIGs, where XOR gates are allowed in addition to ANDs.

B. Boolean Satisfiability (SAT) Problem

The CNF Satisfiability problem (CNF-SAT) [17] 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, i.e., each clause evaluates to one under given assignment of variables.

The decision SAT problem just gives an answer (positive or negative) about satisfiability, whereas its constructive version returns a satisfiability witness, i.e., the satisfying assignment of variables as a result.

C. Pseudo-Boolean Optimization (PBO) Problem

The Pseudo-Boolean Optimization (PBO) problem [18] can be simply understood as a special case of the Integer Linear Programming problem, or an extension of SAT, with optimization capabilities. Instead of processing a product of clauses, a set of linear inequalities is processed. An integer-weighted sum of literals is present at the left-hand side of each inequality and integers are at the right-hand side. Generally, each PBO inequality is written in a form

$$C_0 y_0 + C_1 y_1 + \cdots + C_{n-1} y_{n-1} \geq C,$$

where y_i are Boolean variables and C_i and C are integer constants.

The optimization criterion is defined as an integer-weighted sum of variables, i.e., similarly to the left-hand side of the inequalities, which is to be either minimized or maximized.

Dedicated PBO solvers exist [19], [20], [21], or the problem is solved by repeated application of a SAT-solver [22], [23].

In later sections we will need to transform a standard CNF to a PBO instance. This can be accomplished in a straightforward way:

1) For each variable $x_1, \ldots, x_m$ of the CNF, construct a variable $y_1, \ldots, y_m$ of the PBO.
2) For each CNF clause $(l_1 \vee l_2 \vee \ldots \vee l_j)$, where l_i are its individual literals (variables or their negations), construct an inequality $L_1 + L_2 + \cdots + L_j \geq 1$.
3) If a literal $l_i = x_k$ (variable in its direct form), substitute $L_i = y_k$ in the inequality. If a literal $l_i = \overline{x_k}$ (variable in its negated form), substitute $L_i = (1 - y_k)$.

As an example, let us have a clause $(x_1 \vee \overline{x_2} \vee \overline{x_3})$. This clause can be transformed to a PBO inequality as follows:

$$y_1 + (1 - y_2) + (1 - y_3) \geq 1$$

which is

$$y_1 - y_2 - y_3 \geq -1$$

IV. THE PROPOSED METHOD

Basic principles of the proposed method generating optimum implementations of *k*-variable Boolean functions will be described here. First, we will present it in its basic, SAT-based form, to devise optimal AIG implementation of a given Boolean function.

It will be then extended to support additional features, namely XOR gates, enumeration of all solutions, delay-optimal implementations generation, and customizable gates costs.

A. Basic Procedure

The basic procedure is based on converting the Minimum Circuit problem [24] to the Satisfiability (SAT) problem. The problem can be further simplified to finding *any* implementation of a given *k*-variable function using exactly *n* gates [13].

Therefore, we start with trying to find an implementation of a given function with $n = 1$, i.e., one gate (trivial cases are solved in the pre-processing phase). If no such implementation is found, i.e., the respective SAT instance is unsatisfiable, *n* is increased by one. This is performed until a satisfiable solution is found. As a result, the obtained implementation naturally has the minimum number of nodes.

The input to the algorithm is a truth table (binary vector) of the function to be implemented, the output is its optimal implementation structure. The algorithm is depicted in a pseudo-code in Figure 1:

```
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;
}
```

Figure 1. The optimal structure generation procedure

B. CNF Generation

The main procedure of the algorithm, the SAT instance generation (`Generate_CNF` in Figure 1) is described here.

We assume the AIG [15] circuit implementation in this section. Thus, the circuit consists of AND nodes only, while nodes connections (i.e., AND nodes inputs) may be negated. Also the output can be negated.

These assumptions are considered in the following text:

- Each primary input (PI) and AIG node has a unique index, $0 \ldots n + k - 1$, where AIG PIs are represented by nodes indexed $0 \ldots k - 1$ and AND nodes are those with indexes $k \ldots n + k - 1$,

- the AIG has one output, which is the node with index $n+k-1$;
- the parent node always has higher index than both its children;
- the left child of a node has lower index than the right child of that node;
- node inputs (exactly two here) are labeled 0 (left input) and 1 (right input);

In this scenario, we define a set of variables:

- N_{im} ... m^{th} input of node i is negated, defined for $i \in \{k, \dots, n+k-1\}, m \in \{0,1\}$;
- C_{ijm} ... output of i^{th} node is connected to m^{th} input of j^{th} node, defined for $i \in \{0, \dots, n+k-1\}$, $j \in \{\max(i+1, k), \dots, n+k-1\}, m \in \{0,1\}$
- ON ... output node is negated.

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

1) Each node input is connected somewhere - to a node with a smaller index:

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

2) Each node output is connected somewhere - to a node with a higher index. The AIG output is not assumed:

$$\forall i \in \{k, \dots, n+k-2\}: \bigvee_{\substack{j \in \{\max(i+1,k),\dots,n+k-1\} \\ m \in \{0,1\}}} C_{ijm}$$

3) For each node, the left child has lower index than the right child:

$\forall i \in \{k, \dots, n+k-1\}, j \in \{0, \dots, i-1\}, h \in \{0, \dots, j\}$:
$C_{ji0} \Longrightarrow \overline{C_{hi1}}$

This is, in CNF:
$\forall i \in \{k, \dots, n+k-1\}, j \in \{0, \dots, i-1\}, h \in \{0, \dots, j\}$:
$\overline{C_{ji0}} \vee \overline{C_{hi1}}$

4) Each node input has only one source

$\forall i \in \{k, \dots, n+k-1\}, j \in \{0, \dots, i-1\}$,
$h \in \{j+1; i-1\}, m \in \{0,1\}: C_{jim} \Longrightarrow \overline{C_{him}}$

This is, in CNF:
$\forall i \in \{k, \dots, n+k-1\}, j \in \{0, \dots, i-1\}$,
$h \in \{j+1; i-1\}, m \in \{0,1\}: \overline{C_{jim}} \vee \overline{C_{him}}$

These CNF clauses ensure the correctness of the produced AIG and prevent unnecessary ambiguities, like, e.g., children swapping.

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 this purpose we define additional variables, specific for each $p \in \{0, \dots, 2^k - 1\}$ minterm:

- Y_{ip} ... i^{th} node output value, $i \in \{0, \dots n+k-1\}$. For $i < k$, it represents a primary input (PI).
- X_{imp} ... m^{th} input of node i; defined for $i \in \{k, \dots, n+k-1\}, m \in \{0,1\}$;
- XN_{imp} ... m^{th} input of node i, behind possibly negated edge; defined for $i \in \{k, \dots, n+k-1\}, m \in \{0,1\}$;
- ON_p ... output, after possible negation

Next, clauses enforcing the function are created:

5) Inverted edges:

$$\forall i \in \{k, \dots, n+k-1\}, m \in \{0,1\}: XN_{imp} = X_{imp} \oplus N_{im}$$

This is, in CNF:
$\forall i \in \{k, \dots, n+k-1\}, m \in \{0,1\}: (XN_{imp} \vee \overline{X_{imp}} \vee N_{im})(XN_{imp} \vee X_{imp} \vee \overline{N_{im}})(\overline{XN_{imp}} \vee \overline{X_{imp}} \vee \overline{N_{im}})(\overline{XN_{imp}} \vee X_{imp} \vee N_{im})$

6) Nodes interconnections:

$\forall i \in \{0, \dots, n+k-2\}, j \in \{\max(i+1; k), \dots, n+k-1\}$,
$m \in \{0,1\}: C_{ijm} \Leftrightarrow Y_{ip} = X_{jmp}$

This is, in CNF:
$\forall i \in \{0, \dots, n+k-2\}\, j \in \{\max(i+1; k), \dots, n+k-1\}$,
$m \in \{0,1\}: (\overline{C_{ijm}} \vee Y_{ip} \vee \overline{X_{jmp}})(\overline{C_{ijm}} \vee \overline{Y_{ip}} \vee X_{jmp})$

7) Output negation:

$$ON_p = Y_{(n+k-1)p} \oplus ON$$

This is, in CNF:
$(ON_p \vee \overline{Y_{(n+k-1)p}} \vee ON)(ON_p \vee Y_{(n+k-1)p} \vee \overline{ON}) \cdot (\overline{ON_p} \vee Y_{(n+k-1)p} \vee ON)(\overline{ON_p} \vee \overline{Y_{(n+k-1)p}} \vee \overline{ON})$

8) Nodes function:

$$\forall i \in \{k, \dots, n+k-1\}: Y_{ip} = XN_{i0p} \cdot XN_{i1p}$$

This is, in CNF:
$\forall i \in \{k, \dots, n+k-1\}: (\overline{Y_{ip}} \vee XN_{i0p})(\overline{Y_{ip}} \vee XN_{i1p})(Y_{ip} \vee \overline{XN_{i0p}} \vee \overline{XN_{i1p}})$

9) Network function:

$$\forall i \in \{0, \dots k-1\}: Y_{ip} = \text{forced respective bit value}$$

$$ON_p = \text{forced output for minterm } p$$

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_{im}, C_{ijm}, and *ON*, then represents the implementation of the desired AIG.

Note that the number of clauses describing the AIG validity grows linearly with both *k* and *n*, but the number of clauses describing the function grows exponentially with *k*. Therefore, it is clear that this approach is feasible for small *k*'s only. However, it is fully sufficient for our purposes.

C. Extension to XOR Gates Support

In order to extend the method to support XOR gates apart from ANDs, only very slight modifications need to be done.

The set of variables is extended by only one for each node, indicating whether the respective node is AND or XOR. Thus, such variables are added:

$XOR_i \ldots i^{th}$node is XOR, $i \in \{k, \ldots n+k-1\}$

Next, the clauses describing the node function – see *8)* – are modified in this way:

$\forall i \in \{k, \ldots, n+k-1\}: Y_{ip} = (\overline{XOR_\iota} \Rightarrow XN_{i0p} \cdot XN_{i1p}) \cdot (XOR_i \Rightarrow XN_{i0p} \oplus XN_{i1p})$

This is, in CNF:
$\forall i \in \{k, \ldots, n+k-1\}: \left(XOR_i + \overline{Y_{\iota p}} + XN_{i0p}\right)\left(XOR_i \vee \overline{Y_{\iota p}} \vee XN_{i1p}\right)\left(XOR_i \vee Y_{ip} \vee \overline{XN_{\iota 0p}} \vee \overline{XN_{\iota 1p}}\right)\left(\overline{XOR_\iota} \vee Y_{ip} \vee \overline{XN_{\iota 0p}} \vee XN_{i1p}\right)\left(\overline{XOR_\iota} \vee Y_{ip} \vee XN_{i0p} \vee \overline{XN_{\iota 1p}}\right)\left(\overline{XOR_\iota} \vee \overline{Y_{\iota p}} \vee \overline{XN_{\iota 0p}} \vee \overline{XN_{\iota 1p}}\right)\left(\overline{XOR_\iota} \vee \overline{Y_{\iota p}} \vee XN_{i0p} \vee XN_{i1p}\right)$

Additionally, since this symmetry relation holds:

$\bar{a} \oplus b = a \oplus \bar{b}$,

we may add additional constraints, allowing only the left XOR node negated:

$\forall i \in \{k; n+k-1\}: \overline{XOR_\iota} \vee \overline{N_{\iota 1}}$

This constraint may seem to be unnecessary for XAIG validity; however, it is highly beneficial for purposes of enumeration (see Subsection IV.D), where it prevents generation of many unnecessary symmetries.

Note that extending the method to support other simple 2-input gates (OR, NAND, NOR) is as straightforward as the XOR extension. Only when multiple (more than two) inputs to a gate are to be assumed, the node function variables cannot be binary. Similarly, extension to more-input gates (like the majority function) would involve more complex indexing of node parents.

D. Enumeration

In order to obtain multiple or even all solutions, i.e., all (X)AIG structures implementing the given function, simply an All-SAT solver can be used. However, this approach is not practical, since structurally equivalent solutions with just permuted node indexes would be produced. Therefore, we propose the procedure shown in Figure 2.

Here, the best *n* (minimum number of nodes by which the function can be implemented) with the initial solution is found first. Then the CNF leading to this solution is constrained, so that such a solution will not be generated by a consequent SAT-solver run. This is done by simply adding one *blocking clause* describing the solution, i.e., the sum of variables N_{im}, C_{ijm}, and *ON* to the CNF, with all variables negated. This is done for *all permutations* of nodes indexes, which describe valid (X)AIGs (i.e., where conditions from Subsection IV.B are satisfied).

Then the SAT-solver is invoked for this CNF instance. This procedure is repeated, until an unsatisfiable solution is produced, indicating that no other feasible solutions exist.

As a result, all feasible non-isomorphic solutions are produced.

```
Generate_all (truth_table f, int k) {
   // find minumum n first
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      Sol = SAT_Solve(CNF);
      if (Sol.unsat) n++;
   }
   // enumerate all solutions
   while (Sol.sat) {
      All.append(Sol.extract_structure);
      for_all_feasible_permutations P {
         CNF.Constrain(P, Sol);
      }
      Sol = SAT_Solve(CNF);
   }
   return All;
}
```

Figure 2. All optimal structures generation procedure

E. Delay-Optimum Implementations

Even though the procedures described above produce optimal implementations in terms of nodes count, they need not be delay optimal. The approach presented in [13] uses SMT to evaluate the level of each node and optimize the result according to this. Here the level is expressed as an integer variable attached to each node. However, this would be complicated when a simple SAT-solver is to be used. Therefore, we have decided for an approach, where the enumeration procedure is run (see Subsection IV.D) and only one delay-optimum solution is selected as a result. Naturally, lower bounds on circuit depth can be imposed, thus this solution definitely does not involve enumerating all solutions in most cases.

F. Customizable Gates Costs

The method presented in Subsection IV.C assumed the same cost of AND and XOR nodes. However, this may not always be desired, since XOR gates may be more expensive in general.

Thus, we propose an approach based on Pseudo-Boolean optimization, which can assume higher XOR nodes cost. In its basic form, the implementation is simple and straightforward. The CNF clauses from Subsections IV.B and IV.C are transformed to PBO inequalities, as described in Subsection III.C. Next, the optimization criterion to be minimized is defined as follows:

$$\sum_{i \in \{k, \ldots n+k-1\}} XOR_i$$

A solution of this PBO instance results in an implementation consisted of *n* gates, while AND nodes are preferred over XORs.

This approach, however, cannot be directly used in context of the overall algorithm, see Subsection IV.A. For example, let us assume the XOR cost being 3 and the AND cost 1. A solution with $n = 3$ with one XOR gate has been found. Therefore, the total cost is 5 (two AND gates + one XOR gate). However, we are not sure if there is not a cheaper solution comprised of 4 AND gates.

Also, the *value* of XOR cost is not assumed here at all– XOR is just considered as more costly than AND.

Therefore, the algorithm from Figure 1 must be slightly modified, as shown in Figure 3. The algorithm starts similarly to the original one (Figure 1), only a PBO solver is used instead, in order to incorporate the XOR count minimization. In this phase, the initial solution minimizing the total number of nodes and secondarily the number of XORs is obtained.

Then, the solution cost is computed. This is the place where the XOR cost value comes to play. Note that the AND node cost is assumed to be 1. Thus, the XOR cost is given in multiples of AND node cost.

The second phase of the algorithm tries to find a "cheaper" solution consisted of more nodes. Increasing *n* makes sense while it is lower than the cost of the best solution; in case of equality, the best solution consists of AND gates only.

Also note that the PBO solver needs not be employed in all cases and a (faster) SAT solver can be used instead. This may happen in the second phase, if there is no chance for a better solution containing a XOR gate.

```
Generate_structure (truth_table f, int k) {
   // find initial solution
   n = 1;
   do {
      CNF = Generate_CNF(f, k, n);
      PBO = CNF.ToPBO();
      Sol = PBO_Solve(CNF);
      if (Sol.unsat) n++;
   } while (Sol.unsat);
   Sol.cost = Sol.ANDs +
      Sol.XORs*XOR_cost;
   Best_Sol = Sol;
   // try to find better solution
   while (n < Best_Sol.cost) {
      n++; // increase nodes count
      CNF = Generate_CNF(f, k, n);
      PBO = CNF.ToPBO();
      Sol = PBO_Solve(CNF);
      Sol.cost = Sol.ANDs +
         + Sol.XORs*XOR_cost;
      if (Sol.cost < Best_Sol.cost) {
         Best_Sol = Sol;
      }
   }
   return Best_Sol.extract_structure();
}
```

Figure 3. The optimal structure generation procedure using PBO

The enumeration and delay optimizing versions of the algorithm (see Figure 2) must be modified in a similar way. Thus, starting with the *n* value obtained from the algorithm in Figure 3, *n* must be increased up to the `Best_Sol.cost` value to find all optimum-cost solutions. The difference from the algorithm from Figure 3 is the addition of blocking clauses after each solution found, and repetition of the main loop, while satisfiable solutions are generated, similarly to Figure 2.

V. Experimental Results

Experimental evaluation of the proposed method of optimum structures generation, or particularly statistics for the results of the method will be presented in this section.

Since the properties of optimal AIG implementations are well known, we will mostly focus on XAIG implementations.

The outcomes of the method will be then evaluated in a practical application, the XAIG-based rewriting, in Subsection V.C.

In all experiments, MiniSAT [14] was used as the SAT-solver and MiniSAT+ [23] as the PBO solver.

A. Optimal 4-Input XAIGs

Here we will evaluate the properties of XAIG-based 4-input functions, optimal both in area and delay. The results will be presented for different XOR nodes costs. Thus, results of the delay-optimum XAIG generation method described in Subsection IV.F are presented.

The results are computed from 220 circuits representing non-trivial NPN equivalence classes [3] of all 4-input functions [4].

The results are summarized in TABLE III. For each implementation (AIG just for comparison and XAIG with XOR nodes costs 1, 2, and 3), the maximum and average values, and the sum over all 220 circuits, are presented for the total number of nodes, the number of XOR nodes, and the delay (number of levels).

We can see that with increasing XOR cost, the algorithm tends to generate less XORs. With that, the delay increases. Notice that the *cost* = 3 is so prohibitive, so that the results are approaching the case of AIG.

B. Scalability Evaluation

In order to approximately assess the scalability of the method, we have generated and solved SAT instances for different *k*'s and *n*'s, in terms of the number of CNF variables and clauses. Note that these two parameters strictly determine the CNF size. For the purpose of such complexity estimation, a simple function (NOR of all inputs) was chosen, so that the SAT-solver run-times are negligible. There definitely are functions that will be difficult for the SAT solver. However, these functions are hard to identify a-priori, and thus difficult to find and present. We may only hope that such functions occur in practical circuits only rarely.

TABLE I. Scalability results

k	*n*	Variables	Clauses	Run-time [s]
3	2	131	503	0.01
3	4	237	1 123	0.03
4	2	263	1 134	0.06
4	10	1 071	8 538	0.57
5	10	2003	17 459	2.50
6	10	3 879	36 378	3.70
7	15	11 075	134 945	29.55
10	6	42 147	391 556	369.16
11	20	230 237	3 855 507	21 010.61
12	25	566 499	11 049 756	140 136.85

The results are shown in TABLE I. For given k and n, the numbers of variables and clause are shown, with the algorithm overall run-time. We can see that even though tremendous numbers of variables and clauses are generated for, say, $k = 10$, they are processed very fast.

C. Rewriting Results

As it was stated in the Introduction, the most typical application of optimal functions implementations is the rewriting logic optimization algorithm [1], [6]. In rewriting, a Boolean network (AIG) is traversed in topological order, k-input sub-functions (*cuts*) are extracted from it, and replaced by their (preferably) optimum implementations. This is done iteratively, until no further improvement can be obtained.

The AIG-based rewriting algorithm has been implemented in an academic tool ABC [26]. Recently, we have extended ABC by XAIG-based rewriting [2], [6]. Since the optimal XAIG representations were not available until now, the replacement circuits were generated by logic synthesis from truth tables. Particularly, the replacement circuits were produced by ABC command sequence: "*read_truth; st; dch; if; mfs; st; dch; if; mfs; st; st; dch; if; mfs; st; dch; map*". The technology library used for mapping comprised just inverters, 2-input ANDs, and 2-input XORs. Only 4-input replacement functions (cuts) are used in this algorithm, therefore, those 220 circuits representing non-trivial NPN equivalence classes [4] were synthesized this way.

In our rewriting algorithm implementation, we use only one pre-computed replacement circuit for each NPN-equivalence class. However, all irredundant replacement circuits should be assumed to achieve better results. This will be a part of our future work.

In this subsection we will compare the results achieved by the previous implementation with results obtained by the exact synthesis proposed in this paper.

First, we will compare the qualities of the 220 replacement circuits. Naturally, the optimum implementations always had lower or equal numbers of nodes. The quantitative results are summarized in TABLE II.

TABLE II. REPLACEMENT CIRCUITS COMPARISON

	Non-exact synthesis			Optimum solution		
	Max.	Avg.	Sum	Max.	Avg.	Sum
Nodes	13	6.4	1 341	7	5.0	1 099
XORs	3	0.6	124	5	2.0	436
Levels	6	4.0	834	6	3.2	714

Results of runs of the rewriting algorithm will be presented next. These experiments were performed using a mix of benchmark sets, particularly LGSynth'91 [27], IWLS'93 [28], ISCAS'85 [29], ISCAS'89 [30], and IWLS 2005 [31] - available from [32], 330 circuits were processed altogether.

First, comparison of the original rewriting approach [2], [6] with optimum replacements is given in TABLE IV. For each circuit, the total number of nodes, the number of XOR nodes, and the number of levels are given, for both algorithms. Since the XOR and AND gate costs were assumed equal in the original algorithm, the same is assumed in the optimum replacements. Results for 20 selected circuits (larger circuits with best and worse improvements) are given in the table, while the summary values for all 330 circuits are shown in the last row. The percentage improvements are shown in the last triplet of columns.

We can see that the improvement, in terms of nodes count, is really negligible (0.8%), while there is even deterioration in delay on average. This result is rather unexpected – since the optimum replacement circuits are always equal or better in terms of both area and delay, better rewriting results should be expected. This failure can be attributed to an insufficiency of control of the rewriting algorithm and, probably more significantly, to using only one replacement circuit at a time. For this reason, some suboptimal replacements ended up in better results in some cases, due to better logic sharing possibilities.

What can be clearly seen is that the number of XORs in the result is increased for the optimal replacements. This indeed corresponds with the results from TABLE II.

Finally, rewriting results for different XOR gates costs are presented in TABLE V. We have selected 20 biggest circuits from the set of 330 circuits. For purpose of this comparison, optimum XAIG replacement circuits were generated, with XOR costs 1, 2, and 3. Different XOR costs were also considered in the very rewriting algorithm, when judging about efficiency of the replacement. For details see [6].

We can see that the higher the XOR cost is, the less XORs are in the solution. As a consequence, the network tends to have more levels.

VI. CONCLUSIONS

In this paper we have presented a SAT-based method to compute optimum implementations of Boolean functions of a given number of inputs. In contrast to previously published approaches, it is applicable for generating optimum networks consisted of any type of gates (AND and XOR are chosen for a case study), different gate costs can be assigned, delay-optimum solutions can be obtained, and even all area and delay optimum solutions can be enumerated. All this can be accomplished by using either a standard SAT-solver or a PBO solver when different gates cost are to be considered.

Statistics on the results for XAIG implementations are provided. The effects of adjusting gate costs can be clearly seen from the presented results – the more costly XOR nodes are, the less XORs are used for the implementation.

The scalability of the method was studied for functions of up to 12 variables. We have shown that for "simple" functions the algorithm is very fast. Therefore, we may hope that functions difficult for a SAT-solver occur in practice only rarely, and thus the optimum implementation generation algorithm can be run in synthesis on-line.

Optimal implementations produced by the proposed method are used in a Boolean network optimization algorithm, the rewriting using 4-input functions. The results are compared with the state-of-the-art. Surprisingly enough, no significant improvement has been achieved. Most probably this is due to a lack of high-level control of the rewriting algorithm and to using

only one replacement circuit at a time. Finding a remedy will be part of our future work.

ACKNOWLEDGEMENT

This research has been partially supported by the grant GA16-05179S of the Czech Grant Agency (2016-2018) and by the CTU grant SGS17/213/OHK3/3T/18.

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.

Also let us express many thanks to Alan Mishchenko, for his valuable hints and discussion.

REFERENCES

[1] 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.

[2] I. Halecek, P. Fiser, and J. Schmidt, "Are XORs in logic synthesis really necessary?," in Proc. of IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Dresden (DE), April 19-21, 2017.

[3] S. Muroga, Logic Design and Switching Theory John Wiley & Sons, Ltd, 1979.

[4] J. N. Culliney, M. H. Young, T. Nakagawa, and S. Muroga, "Results of the synthesis of optimal networks of AND and OR gates for four-variable switching functions," IEEE Trans. on Computers, vol. 28, no. 1, pp. 76–85, 1979.

[5] E. Boros, P. L. Hammer, "Pseudo-Boolean Optimization". Discrete Appl. Math., vol. 123, no. 1-3, Nov 2002, pp. 155-225.

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

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

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

[9] 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.

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

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

[12] 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.

[13] M. Soeken et al. "Exact Synthesis of Majority-Inverter Graphs and Its Applications", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, February 2017.

[14] N. Éen, N. Sorensson, "An extensible SAT-solver," in Lecture Notes in Computer, Science 2919 - Theory and Applications of Satisability Testing. Springer Verlag, 2004, pp. 333-336.

[15] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 12, pp. 1377–1394, 2001.

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

[17] 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.

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

[19] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS, "A Backtrack Search Pseudo-Boolean Solver," in Symposium on the Theory and Applications of Satisfiability Testing (SAT), 2002

[20] V. M. Manquinho and O. Roussel, "The First Evaluation of Pseudo-Boolean Solvers," in Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2, pages 97–136, 2006.

[21] H. M. Sheini and K. A. Sakallah, "Pueblo: A Hybrid Pseudo-Boolean SAT Solver," in Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2, pages 157–181, 2006.

[22] O. Bailleux, Y. Boufkhad, and O. Roussel, "A Translation of Pseudo Boolean Constraints to SAT," in Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2, pp. 183–192, 2006.

[23] N. Eén, N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT", in Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, pp. 1-26, Nov. 2006.

[24] V. Kabanets and J. Cai, "Circuit minimization problem," in Symposium on Theory and Computing, 2000, pp. 73–79.

[25] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison-Wesley, 2015.

[26] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification" [Online]. Available: http://www.eecs.berkeley.edu/alanmi/abc/.

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

[28] K. McElvain, "IWLS'93 Benchmark Set: Version 4.0," Tech. Rep., May 1993.

[29] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in IEEE International Symposium Circuits and Systems, 1985, pp. 677–692.

[30] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in IEEE International Symposium on Circuits and Systems, May 1989, pp. 1929–1934 vol.3.

[31] C. Albrecht, "IWLS 2005 benchmarks," Tech. Rep., June 2005.

[32] P. Fiser and J. Schmidt, "A comprehensive set of logic synthesis and optimization examples," in 12th. Int. Workshop on Boolean Problems (IWSBP), 2016, pp. 151–158. [Online]. Available: http://ddd.fit.cvut.cz/prj/Benchmarks/.

TABLE III. COMPARISON OF DIFFERENT XOR NODES COSTS

	AIG (no XORs)			XAIG, XOR cost 1			XAIG, XOR cost 2			XAIG, XOR cost 3		
	Max.	Avg.	Sum	Max.	Avg.	Sum	Max.	Avg.	Sum	Max.	Avg.	Sum
Nodes	10	6.9	1522	7	5.0	1099	7	5.1	1128	10	5.8	1278
XORs	0	0.0	0	5	2.0	436	3	1.2	263	3	0.6	122
Levels	6	3.9	851	6	3.2	714	5	3.4	742	6	3.7	804

TABLE IV. REWRITING RESULTS

	Old approach [6]			Optimum replacement			Improvement	
Circuit	Nodes	XORs	Levels	Nodes	XORs	Levels	Nodes	Levels
Z5xp1	151	8	9	125	20	9	17.22%	0.00%
mult16b	106	30	3	91	45	3	14.15%	0.00%
mult32b	214	61	3	184	91	3	14.02%	0.00%
sqn	108	3	11	93	8	11	13.89%	0.00%
mult32a	219	63	95	189	93	95	13.70%	0.00%
mm30a	1037	116	156	899	172	155	13.31%	0.64%
cht	187	0	6	165	0	6	11.76%	0.00%
mm9a	310	32	51	277	47	49	10.65%	3.92%
s4863	1132	263	46	1016	368	38	10.25%	17.39%
too_large	6285	0	28	5646	1	28	10.17%	0.00%
mm4a	202	0	26	211	3	26	-4.46%	0.00%
spla	1230	19	25	1300	35	25	-5.69%	0.00%
x2dn	186	7	14	199	7	19	-6.99%	-35.71%
br2	143	2	13	153	6	13	-6.99%	0.00%
5xp1	169	2	8	182	7	9	-7.69%	-12.50%
mark1	321	5	27	348	9	24	-8.41%	11.11%
x9dn	153	0	14	174	0	19	-13.73%	-35.71%
vtx1	152	0	12	176	0	19	-15.79%	-58.33%
x1dn	152	0	12	176	0	19	-15.79%	-58.33%
i8	1600	0	21	1897	51	21	-18.56%	0.00%
Sum	**274 127**	**10 954**	**5 939**	**271 938**	**25 627**	**5 978**	**0.80%**	**-0.66%**

TABLE V. REWRITING - COMPARISON OF RESULTS FOR DIFFERENT XOR NODES COSTS

	XOR cost 1			XOR cost 2			XOR cost 3		
Circuit	Nodes	XORs	Levels	Nodes	XORs	Levels	Nodes	XORs	Levels
s6669	1386	462	58	1476	372	56	1476	372	56
c6288	1436	476	75	1467	433	77	2334	0	120
b2	1553	15	20	1541	5	20	1545	4	20
in1	1553	15	20	1541	5	20	1545	4	20
bcd	1809	17	23	1803	11	23	1807	10	23
i8	1897	51	21	1894	0	21	1894	0	21
bcc	2565	29	24	2551	14	24	2556	14	24
bcb	2623	32	25	2623	19	25	2634	16	25
ex1010	2699	132	21	2763	65	22	2897	47	22
dsip	2707	5	10	2709	2	13	2711	1	13
prom2	2845	123	20	2945	57	20	3061	35	20
bca	3086	26	25	3079	18	25	3086	16	25
seq	3665	3	15	3665	0	15	3665	0	15
piir8o	3815	806	37	3925	656	38	4515	357	57
des	3833	96	16	3821	82	16	3764	78	16
bigkey	4224	220	9	4446	6	10	4448	5	10
misex3	4440	2	15	4440	0	15	4440	0	15
piir8	4828	1592	39	5173	1275	41	6169	769	45
too_large	5646	1	28	5645	0	28	5645	0	28
prom1	6323	209	23	6512	89	23	6710	55	23
Sum	**27 1938**	**25 627**	**5 978**	**274 723**	**10 643**	**6 005**	**280 226**	**7 924**	**6 092**