On XAIG Rewriting

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

***Abstract*—This paper presents a rewriting algorithm based on XOR-AND-Inverter Graphs (XAIGs). Such logic representation allows for synthesis algorithms work with XOR gates in a native way. Basic support for XAIGs has already been implemented in the ABC9 package GIA manager, where AND, XOR, and MUX nodes are allowed. However, the rewriting algorithm was missing in ABC9, hence we have re-implemented the original, AIG-based algorithm, and included it in the ABC9 package. For our purposes, we have restricted the nodes set to AND and XOR gates only. The presented XAIG-based rewriting algorithm is compared with the standard AIG-based rewriting algorithm. Since more node types bring some new decisions the rewriting algorithm has to make, we have implemented different rewriting mechanisms and compared them. The results indicate that XAIG based rewriting can lead to different circuits than the original rewriting and although AIG based rewriting produces slightly smaller circuits in most cases, some circuits are better handled with XAIG based rewriting, which points to future research and improvements. The XAIG based rewriting was also able to find significantly more XORs than ABC9 structural hashing and rewriting.**

I. Introduction

Even though the process of logic synthesis and optimization seemed to be an already efficiently resolved problem in past decades, recently there appeared several brand new approaches to it, mostly based on novel data structures used for representing functions and networks.

Already a traditional network representation, And-Inverter-Graphs (AIGs), have been proposed in 2000's [1], [2], [3]. Here a Boolean network (circuit) is represented as a directed acyclic graph (DAG) of 2-input AND nodes, with possibly inverted edges. Numerous algorithms based on AIGs have been developed and implemented in an academic state-of-the-art logic synthesis tool ABC [4]. Most probably, AIGs are (or soon will be) incorporated in commercial tools as well [5]. The authors of these algorithms and tools rightfully claim, that they are scalable [6].

For many years it seemed that AIGs are the ultimate solution to a network representation. However, new and more complex representations emerged recently. Particularly, ABC9 package with a new AIG manager called GIA has been added to ABC by its authors, introducing a possibility to use XOR or MUX nodes in addition to standard AND nodes. This package is however poorly documented and most of synthesis algorithms do not use the GIA manager.

Next, Majority-Inverter-Graphs (MIGs) [7] have been proposed as an alternative to AIGs. Here the two-input AND nodes were replaced by three-input majority functions. MIGs are generalizations of the original structures, thus they are no less scalable. The primary motivation behind introducing these representations was in "emerging technologies" [8]. MIGs were then extended to support XOR gates as well, forming XOR-Majority Graphs (XMGs) [9]. These structures directly reflect the behavior of possibly new technological primitives, thus developing algorithms based on these structures may open new ways of future logic synthesis [8].

Apart from the motivation in emerging technologies, the motivation for introducing more complex structures is in inefficiency of logic synthesis. It has been observed that "standard" structures (network of Sums-of-Products like in SIS [10], AIGs) and algorithms based on them do not efficiently cope with XOR gates [11]. Particularly, most algorithms are based on AND and OR gates and heavily rely on their properties. However, XOR gates are somewhat special. They are difficult to be *identified* in the former structures (SOP, BDD, AIG) and most importantly, the algorithms do not treat them *explicitly*; typically they are identified in the technology mapping phase only. Even though there do exist algorithms performing XOR decomposition [12], [13], [14], no global network processing algorithm assumes XORs explicitly. This may yield in a lack of performance of tools, especially for XOR-intensive circuits [15], [11].

In ABC, bad synthesis performance [15] for XOR-intensive circuits can be either in algorithms inability to utilize XORs and considering AIG as a pure network of ANDs and inverters, or in XOR identification in the network.

Based on ABC internal network representation, we introduce XOR-AND-Inverter Graphs (XAIGs), as an alternative to standard AIGs. Here the Boolean network is represented as a DAG of two types of nodes: 2-input ANDs and 2-input XORs, with possibly inverted edges.

Note that XAIGs represent an orthogonal approach to Majority-Inverter-Graphs (MIGs) [7]. The majority function (as well as AND) is monotonic, while XOR is *not monotonic*. Therefore, XAIGs do cover *all* relevant classes of NPN equivalence [16], [11]. The newly introduced XMGs [9] should have the same property.

To address the issue of algorithms inability to work with XORs natively, we present an XAIGs-based rewriting algorithm [3]. We have implemented it as a command in the ABC9 tool [4]. The experimental results show that XAIG-based rewriting can lead to finding additional XORs in a

network as well as it can help mappers to reduce the area or delay, although it does not provide ultimate solution for all circuits.

The paper is organized as follows: after the Introduction and some preliminaries in Section II, the proposed XAIG structure is described in Section III, with implementation issues presented in Section IV. The newly introduced rewriting algorithm based on the XAIG structure is presented in Section V. Section VI contains experimental results. Section VII concludes the paper.

II. Preliminaries

And-Inverter Graphs (AIGs) [1], [2], [3], are directed acyclic graphs with one or more roots, where nodes are two-input AND gates 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. AIGs are constructed from primary inputs to primary outputs, assigning to each node a unique ID in increasing order. This ensures parent nodes to have higher IDs than their children. The node with lower ID is always the left child of its parent. Apart from that, upon node creation, a hash is calculated from hashes of its children. If a node with the same hash is already present in the graph, this existing node is used by the reference instead of creating a new node. This process is called structural hashing (*"strashing"*) [1] and ensures that there will be no structurally equivalent subgraphs in an AIG.

Structural hashing still does not discover functionally equivalent subgraphs with different structures. ABC provides functionally reduced AIGs, FRAIGs [17]. If this approach (colloquially called "fraiging") is used in addition to structural hashing, also functional hashing of small subgraphs is performed.

A cut of a node *N* is a set of nodes (called *leaves*), for which it holds that every path from primary inputs to the node *N* leads through at least one *leaf*. A cut is *K-feasible* if the number of leaves does not exceed *K*.

III. The XAIG Structure

We propose an extension of AIGs in this paper, the XOR-AND-Inverter graphs (XAIGs). XAIG is a directed acyclic graph, where nodes are two-input ANDs or XORs, while edges can be inverted. Basically, XAIGs represent a special case of AND/XOR/MUX Graphs, which are already implemented in the ABC9 package. As seen in Figure 1, XOR is described by at least 3 AND nodes in AIG, which can make it more difficult for algorithms to utilize it. In XAIGs, as well as in AND/XOR/MUX Graphs, XOR is represented as a single node.

A. XAIG Properties

XAIGs are a generalized form of AIGs; every AIG can be considered as an XAIG with no XOR nodes. Therefore, XAIG (just like AIG) can implement any logic function.

XAIG is structurally hashed in the same way as AIG, with different hashing function for XOR nodes. The fraiging technique is applicable also for XAIGs.

IV. Implementation of XAIGs in ABC

The ABC9 package features a new manager for AIG manipulation, called GIA manager. It has a possibility to use XOR and MUX nodes in addition to standard AND nodes. Therefore, we used this package and GIA as a manager for XAIGs. A network can be converted between the managers by the command *&get* to convert it from the original AIG manager to GIA and *&put* to convert it back.

A. The XIAG File Format

For the need of storing XAIGs in a file with unchanged structure, we defined the XAIGER file format based on the AIGER format [18] and implemented its support to ABC9 network reading and writing commands, *&r* and *&w*. The header of XIAG is described as follows:

xaig M I L O A X, where

- $M = I + L + A + X$,
- *I* stands for the number of inputs,
- *L* stands for the number of latches,
- *O* stands for the number of outputs,
- *A* stands for the number of ANDs,
- and *X* stands for the number of XORs.

The nodes themselves are defined in the same way as in the original AIGER, XORs are distinguished from ANDs by having the left child node ID higher than the right one, which is forbidden in the original AIG. As an example, XOR in the XAIGER format can be described in the following way:

```
xaig 3 2 0 1 0 1
2
4
6
6 2 4
```

This format seems to be no more necessary, as we found out that if XAIG is converted to the original AIG manager with the command *&put* and then stored to a BLIF file, it can be later reconstructed by loading and converting back to GIA by XOR-supporting structural hashing (ABC command *&st*). However, one can never be sure that the structural hashing will reconstruct the original XAIG structure exactly.

B. Recognizing XOR Gates

XOR identification is performed in the ABC9 package by structural hashing (command *&st -m -L 1*). While the parameter *-m* enables conversion to "large" gates (XOR, MUX), the parameter -L sets the reference limit for enabling generation of MUX nodes. We have implemented a new feature in ABC9, where setting this limit to 1 disables MUX creation completely.

This procedure identifies XOR gates represented by 3 ANDs as described in Figure 1. If XOR is dissolved to a flatter structure (i.e., XOR of another functions expressed as a sum of products), the *&st* command is too weak to identify it and

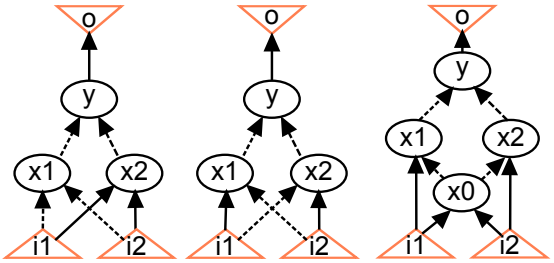

Fig. 1: XOR structures in AIG. Two leftmost structures, composed from 3 ANDs, can be identified by ABC9 structural hashing. The rightmost graph represents the simplest XOR structure not identified by structural hashing in ABC9.

synthesis will be unable to use it, unless it finds it by a different way, e.g., by a functional checking of a subtree, which we implemented in our *&rewrite* command.

V. XAIG-Based Rewriting Algorithm

Listing 1: Rewrite over XAIG network

```
Rewriting (network XAIG, node startNode, hash table
    PrecomputedStructures, bool dissolveXORs)
{
  for each node N topologically ordered after
      startNode in the network XAIG
  {
    bestXAIG = NULL; BestGain = -1;
    for each 4-input cut C of node N computed using
        cut enumeration {
      F = Boolean function of N in terms of the
          leaves of C
      // get best cut implementation
      bestCircuit =
          HashTableLookup(PrecomputedStructures, F);
      // get XAIG with cut replaced by best circuit
      for useRealXORs in (true[, false if
          dissolveXORs==true]){
          rwrXAIG = ReplaceCutByBestCircuit(XAIG,
              bestCircuit, cut, useRealXORs);
          Gain = NodesCost(XAIG) -
              NodesCost(rwrXAIG);
          // keep track of best possible rewrite
              for current node
          if (Gain > 0){
            if (bestXAIG = NULL || BestGain < Gain){
              bestXAIG = rwrXAIG; BestGain = Gain;
            }
          }
      }
    }
    if (bestXAIG != NULL){
        return Rewriting(bestXAIG, N+1,
            PrecomputedStructures, dissolveXORs);
    }
    else {
        continue;
    }
    return XAIG;
  }
}
```

To demonstrate whether synthesis needs to be capable of a native work with XOR gates, we introduce an XAIG rewriting algorithm based on AIG rewriting technique presented in [3].

Rewriting is a technique of replacing AIG subgraphs with k leaves (k-feasible cuts [19]) by smaller, functionally equivalent precomputed structures. This algorithm can reduce functionally equivalent subgraphs, unlike structural hashing can. An example of one subgraph replacement can be seen in Figure 2.

A. The Basic Rewriting Algorithm

As described in Listing 1, the newly introduced algorithm *&rewrite* goes through XAIG nodes in topological order from defined starting node. For each node, cuts are enumerated using the algorithm presented in [19], described in Listing 2. For each node cut, a truth table of the function of its leaves is calculated by simulation. This truth table is then converted to a canonical form described by an integer value, which is stored in a precomputed table for each possible function, so are the permutations of inputs and negations of inputs and outputs needed for this conversion (NPN equivalence classes). Conversion to the canoncial form is done by the same conversion table already available for the original rewriting. For every truth table in canonical form, there is a precomputed "best structure". The summary cost of all nodes is compared for the network before and after replacement of a cut by this structure. Replacement with the highest cost saved is performed for each node and rewriting continues with the next node in topological order.

This can not only reduce the number of nodes for a particular cut, but as all cuts representing the same function are replaced by the same structure, also new sharing can be found in whole network, by structural hashing.

Apart from using XOR nodes in addition to AND nodes and different "best structures", we expect no significant differences from the original *rewrite* algorithm behaviour.

Note that for 4-feasible cuts, there are 2^{16} possible functions, but every possible cut can be converted by permutation of inputs and negation of inputs and output to one of 222 NPN equivalent classes [3]. Therefore, using 4-feasible cuts for rewriting is a good compromise between the efficiency of the algorithm and its memory demands. 4-feasible cuts are also used in original abc command *rewrite*.

The total network cost is calculated as a weighted sum of nodes costs, where for each node type (AND and XOR) a cost is specified. The cost can be adjusted with respect to expected target technology. For example, when FPGA (LUT) mapping is targeted, the same cost for both node types can be set, while for gate-level library targeted, the cost of 2 for AND and 5 for XOR node can be set, to reflect different sizes of the gates. The default cost is 1 for both AND and XOR, and it can be changed by the *&rewrite* parameter *-c*.

B. Rewriting Options

The presence of XOR gates in the rewriting process imposes additional possibilities of choice. Particularly, XOR gates, either present in the original XAIG or newly introduced by cut replacement, may or may not be "dissolved" into three

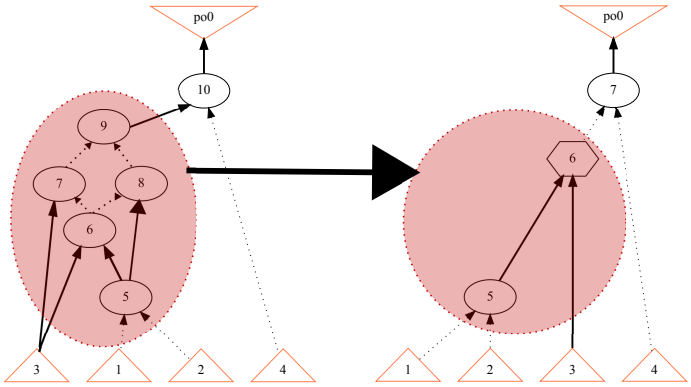

Fig. 2: XAIG based rewriting example. Circle nodes represent AND nodes, hexagon a XOR node. Dotted edges are inverted.

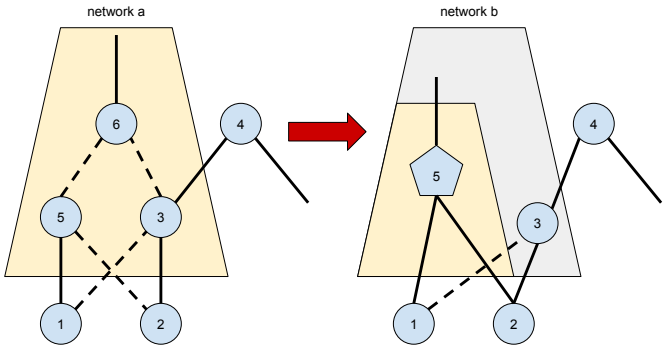

Fig. 3: Duplication of AND nodes after cut replacement

AND gates, depending on sharing possibilities. Therefore, we have implemented an option to dissolve XORs, which leads the algorithm to perform two replacement alternatives for each rewriting step. In addition to the standard replacement using 1-node XORs, XOR nodes are dissolved into three AND nodes as it would be in the original AIG based rewriting. These two alternatives are compared and the one with lower total cost is used. The "dissolving" can happen especially when a non-fanout-free XOR is being replaced and one or both inner AND nodes of the XOR function would have to be duplicated to preserve inputs for nodes outside the cut. Note that if the alternative without a 1-node XOR is selected, the algorithm with this option can still produce 1-node XORs in other rewriting steps.

An example of this situation can be seen in Figure 3. A XOR function has been found inside the cut (network a), but one of its inner nodes (*3*) has an edge which leads outside the cut. This part of the cut can either be replaced by a XOR node and the inner AND node (*3*) must be duplicated to preserve the input to the node *4* (network b). Other option is to preserve the XOR representation by 3 ANDs, so the node *3* does not need to be duplicated. The inner node might still have to be duplicated in case that the best circuit found for this cut would have a different structure, where a node with the same function as the node *3* would not be present elsewhere.

If a flag *-f* is specified in *&rewrite*, each cut is checked for edges leading outside of it before rewriting, and if any is found, this cut is not considered for rewriting (i.e., only fanout-free cuts are considered). This option was implemented for experimental purposes and leads to significantly worse results.

Listing 2: Cut enumeration algorithm

```
void NetworkKFeasibleCuts (Graph g, int k){
  for each node n of g {
    NodeKFeasibleCuts(n, k);
  }
}

cutset NodeKFeasibleCuts (Node n, int k){
  if (n is primary input) return {{n}};
  if (n is visited) return NodeReadCutSet(n);
  mark n as visited;
  cutset Set1 =
      NodeKFeasibleCuts(NodeReadChild1(n), k);
  cutset Set2 =
      NodeKFeasibleCuts(NodeReadChild2(n), k);
  cutset Result = MergeCutSets(Set1, Set2, k) U {n};
  NodeWriteCutSet(n, Result);
  return Result;
}

cutset MergeCutSets (cutset Set1, cutset Set2, int
    k){
  cutset Result = {};
  for each cut Cut1 in Set1{
    for each cut Cut2 in Set2{
      if (|Cut1 U Cut2| <= k){
        Result = Result U {Cut1 U Cut2};
      }
    }
  }
  result Result;
}
```

C. Best logic structures generation

All NPN equivalent classes of cuts mentioned above have already been enumerated in ABC for the original *rewrite* algorithm. The XAIG based rewriting algorithm however needs different "best structures" than the original AIG rewriting, i.e., with XOR nodes allowed. For every NPN class, we calculated this structure by the following sequence of ABC commands: *read_truth; st; dch; if; mfs; st; dch; if; mfs; st; st; dch; if; mfs; st; dch; map; write_blif* to optimize the initial representation described by a truth table and map it using a custom library providing ANDs, XORs, and inverters. ANDs and XORs have the same cost during the mapping to not prefer any gate. The final sequence of commands just converts the mapped structure generated for each NPN class to the XAIGER format to be used with the *&rewrite* command: *read_blif; &get -m; &st -m; &w*.

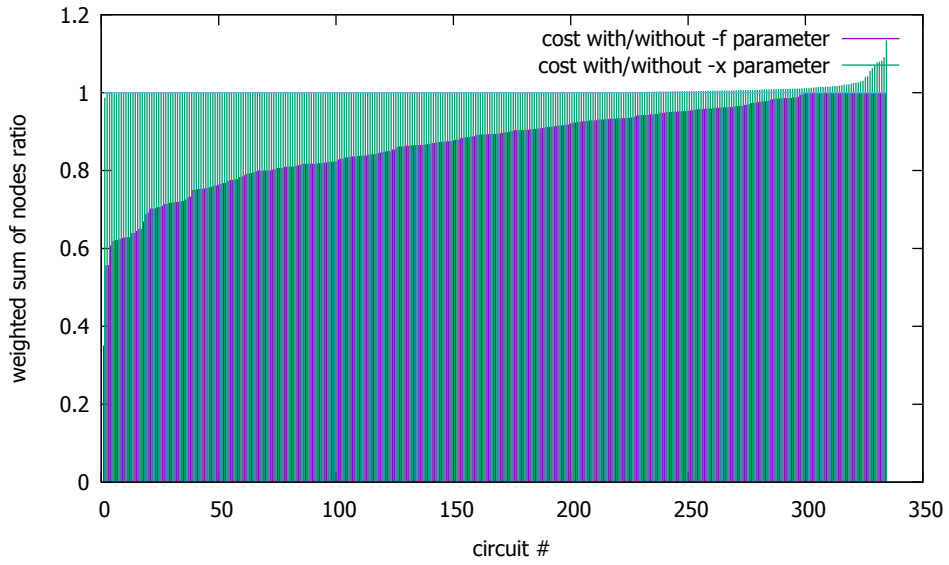

Fig. 4: Comparison of different *&rewrite* configurations in means of weighed sum of nodes, which was set to 2 for AND and 5 for XOR. Values below 1 indicate a better result with respective parameter enabled.

VI. Experimental Results

The experiments were conducted using circuits from a mix of benchmark sets: LGSynth'91 [20], IWLS'93 [21], ISCAS'85 [22], ISCAS'89 [23], Advanced Synthesis Cookbook [24], IWLS 2005 [25] and others [26], [27], and [28] - available from [29].

Altogether, more than 1 400 circuits were processed. However, for some experiments only their limited subset was used, due to time-consuming measurements.

A. Evaluation of Rewriting Options

Figure 4 shows the influence of the *-x* (dissolving XORs) and *-f* (allowing fanout-free cuts only) parameters on the total network cost for the *&rewrite* algorithm. The histogram has been generated by sorting the resulting circuits by cost ratio in increasing order. Values lower than 1 mean better results without using the respective option. The graph is composed of vertical lines, one for each circuit, with the length of the corresponding cost ratio, sorted in ascending order. As for using fanout-free cuts only (*-f*), this histogram shows that this produces much worse results. When dissolving of XOR nodes is enabled (*-x*), results are better for the majority of circuits, although for some circuits this option leads to slightly worse results. Therefore, the option *-x* will be used for comparison with AIG based rewriting, to better demonstrate its strength.

B. Comparison of AIG- and XAIG-Based Rewriting Algorithms

To measure the influence of native XOR nodes on the rewriting process, we compared XAIG based rewriting to the original AIG based rewriting by total cost of the network. The cost of AND nodes was set to 2 and 5 for XOR nodes. The comparison can be seen in Figure 5. However, this is not an ultimate comparison of the algorithms as different node costs would lead to different results.

C. XOR Nodes Introduced by XAIG-Based Rewriting

Table I shows a comparison of the numbers of total nodes and XOR nodes found by *&rewrite* with different values of the nodes cost parameter *-c*. We also included the same information for the original circuit and *&strashed* AIG based rewriting.

XAIG based rewriting is able to find more XORs than *&st* could find after AIG based rewriting. The number of XOR nodes revealed decreases with increasing AND:XOR cost ratio. This also naturally affects the total number of nodes of the final network – the more XORs have been found, the less nodes the final network has.

D. The Overall Synthesis Process

Here the XAIG based rewriting was compared to AIG rewriting in the overall synthesis process. The network optimized by rewriting was mapped to FPGA LUTs and the numbers of LUTs and levels were compared. The AIG rewriting process has been performed by ABC commands *rewrite; &lf; mfs* while the XAIG rewriting process by *&rewrite; &lf; mfs*. The algorithms performance has been measured over 1434 circuits from the sets mentioned above. Table II shows 10 selected circuits from the set, where *&rewrite* produced best results in term of the number of LUTs compared to AIG based rewriting, and 10 circuits showing the same for the number of mapped network levels. The number of LUTs has been decreased by *&rewrite* in 185 cases, while it was increased in 471 cases, other circuits ended up with the same number of LUTs for both algorithms. The level was decreased by *&rewrite* in 144 cases, while in 47 cases it was increased.

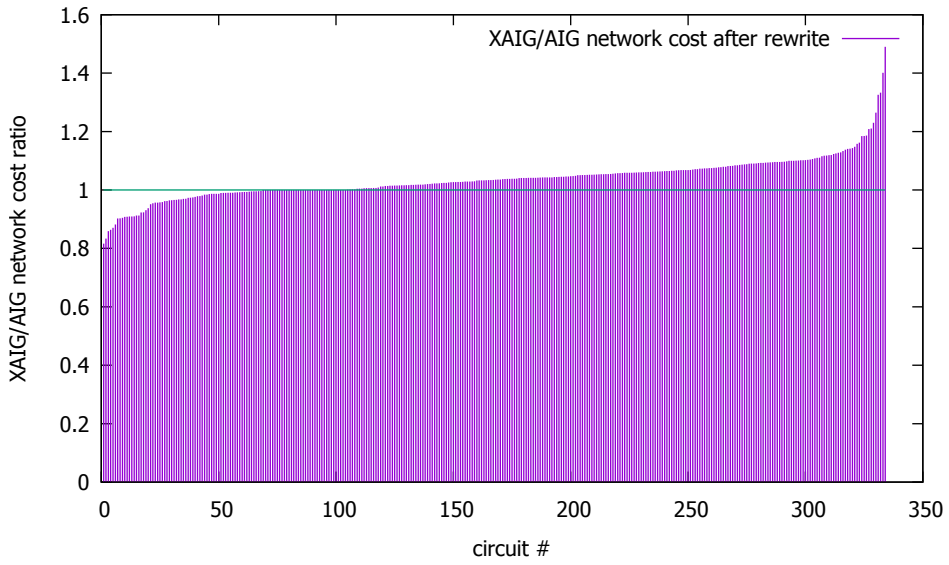

Fig. 5: Comparison of AIG-based rewriting with XAIG-based rewriting by the total network cost. The nodes costs were set to 2 for AND and 5 for XOR. Values below 1 indicate a better result for XAIG based rewriting.

TABLE I: Comparison of XORs found by AIG and XAIG based rewriting. XAIG rewriting has been performed with three different AND:XOR cost ratios. Summary numbers of nodes and XORs for all 335 circuits are shown in the row *total*. The last two rows show the number of cases where the final network had more nodes and where more XORs have been found than in AIG based rewriting with the *&st* command applied at the end and the number of cases where AIG rewriting resulted in higher number of the same stats.

	original		&rewrite 1:1		&rewrite 1:3		&rewrite 2:5		rewrite; &st	
name	nodes	xor	nodes	xor	nodes	xor	nodes	xor	nodes	xor
c6288	2337	0	960	476	2334	0	1034	433	2249	28
bigkey	5134	5	3934	109	4252	4	4252	4	4140	5
mm30a	1396	60	921	116	1119	2	1035	30	889	58
c1355	510	0	100	107	174	82	104	104	278	56
prom2	3461	27	2904	67	2989	34	2946	46	2887	32
s635	190	0	96	31	162	0	156	2	160	0
i6	402	0	337	28	395	0	395	0	387	0
s5378	1283	23	948	82	986	65	965	72	1066	56
g25	170	50	140	50	178	27	178	27	167	30
ex1010	3250	43	2784	62	2835	41	2805	51	2670	46
mm9a	418	18	278	32	329	2	306	10	268	16
Altera_oc_hdlc	2478	132	1665	149	1757	117	1680	141	1672	134
Mentor_1_11	2720	3	2073	62	2105	50	2084	56	2113	47
Mentor_1_12	2720	3	2073	62	2105	50	2084	56	2113	47
mm9b	481	19	318	32	385	3	361	11	326	17
total	**263450**	**5289**	**214989**	**6632**	**217966**	**4588**	**215846**	**5629**	**207579**	**5345**
more than AIG			60	123	33	25	221	56		
less than AIG			210	19	254	146	44	102		

The results can also be seen in Figure 6. The histogram has been generated by sorting the results by the ratio of the number of LUTs and levels after AIG based synthesis to XAIG based synthesis. Circuits, where XAIG based synthesis produced better results are situated on the left, while circuits with worse results on the right.

We can see from these results, that in many cases the XAIG-based rewriting enabled a more delay-efficient implementation.

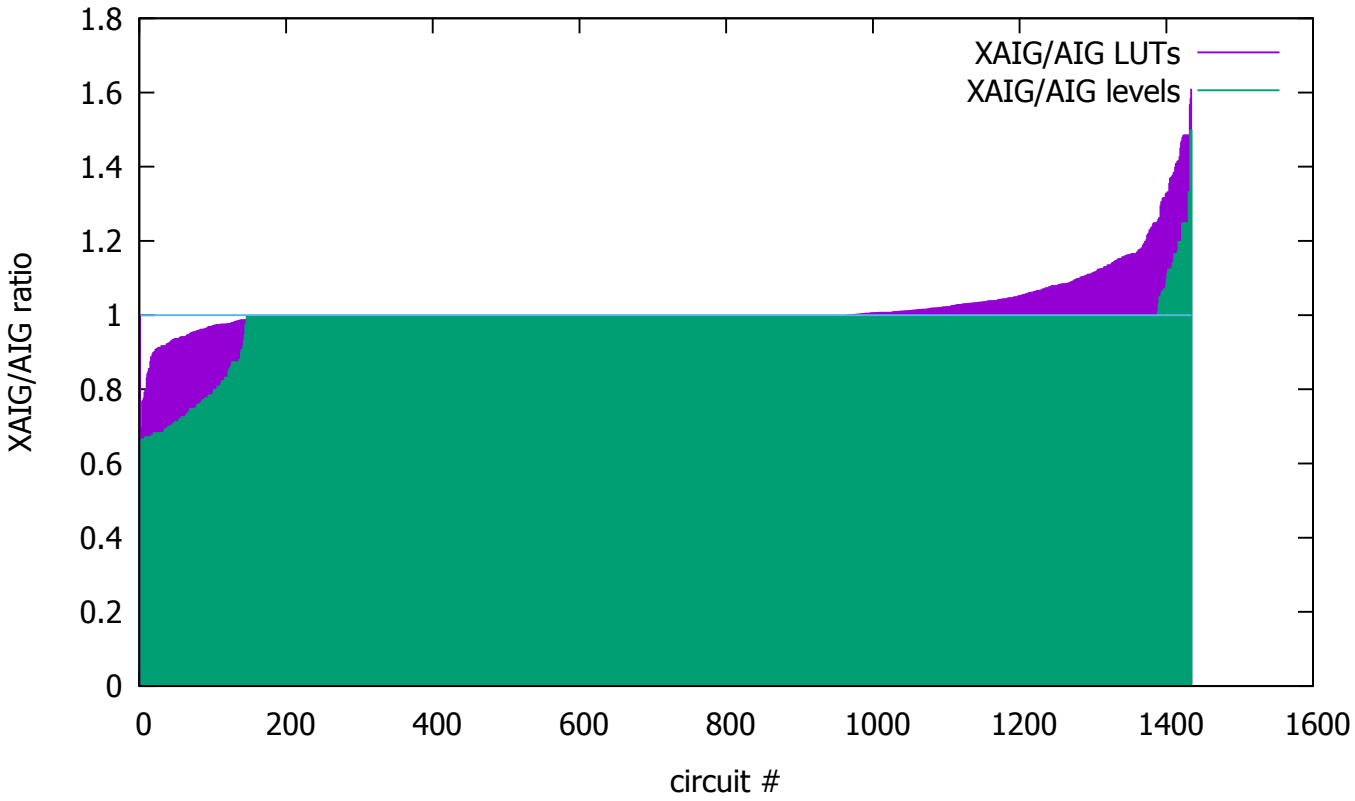

Fig. 6: Comparison of AIG-based rewriting to XAIG-based rewriting by the number of LUTs and levels. Values below 1 indicate a better result for XAIG-based rewriting.

TABLE II: XAIG vs. AIG based rewriting compared after LUT mapping. 10 circuits where XAIG based rewriting produced best results in terms of the number of LUTs compared to AIG rewriting are shown, as well as 10 circuits with worst XAIG based rewriting results.

	rewrite; &lf; mfs		&rewrite; &lf; mfs		XAIG : AIG ratio	
name	LUTs	levels	LUTs	levels	LUTs	levels
uoft_raytracer [26]	9	2	6	2	0.67	1.00
bbtas [27]	10	2	7	2	0.70	1.00
six_three_comp [24]	13	3	10	3	0.77	1.00
mux [21]	27	6	21	5	0.78	0.83
cordic [20]	29	5	23	4	0.79	0.80
life [28]	30	5	24	5	0.80	1.00
majority [21]	5	2	4	2	0.80	1.00
cordic [21]	471	9	392	9	0.83	1.00
term1 [20]	79	6	67	6	0.85	1.00
prio_encode [24]	7	2	6	2	0.86	1.00
cht [21]	51	3	46	2	0.90	0.67
soft_ecc_ram_64bit [24]	61	3	62	2	1.02	0.67
mult16 [30]	131	18	144	13	1.10	0.72
sct [20]	28	4	29	3	1.04	0.75
s400 [21]	45	4	50	3	1.11	0.75
s444 [21]	44	4	50	3	1.14	0.75
s4863 [23]	24	4	28	3	1.17	0.75
lal [21]	27	4	34	3	1.26	0.75
cordic [20]	29	5	23	4	0.79	0.80
vga_driver [24]	121	5	122	4	1.01	0.80

VII. Conclusion

We have implemented an XAIG based rewriting algorithm in the framework of logic synthesis and optimization tool ABC. This variant of rewriting algorithm was compared to the original AIG-based rewriting already implemented in ABC. The results indicate that the new algorithm is stronger in XOR identification than XOR-aware structural hashing, already implemented as a command *&st* in ABC. The XAIG based rewriting can also be beneficial in many cases in terms of delay-efficiency when used in synthesis process.

XOR nodes in XAIG based synthesis bring new decisions, which need to be done, for example whether to create a XOR node even at expense of adding additional AND nodes. XOR can also have different, target technology dependent, cost than AND, i.e. it might be beneficial to have multiple AND nodes instead of one XOR node in the network. These decisions are configurable through *&rewrite* parameters and their influence to the final network structure has been examined.

While the XAIG based rewriting does not bring an ultimately better process than the original AIG based rewriting, the results point out an interesting area of future research on rewriting efficiency. Our future work will be focused on extension of *&rewrite* to use more than one "best structure" per cut function, as well as "best structures" generator based on exact synthesis.

Acknowledgment

This research has been 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) and by the 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," 2012. [Online]. Available: http://www.eecs.berkeley.edu/˜alanmi/abc

[5] R. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–40.

[6] K. Brayton, Robert and A. Mishchenko, "Scalable logic synthesis using a simple circuit structure," in *International Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[7] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Boolean logic optimization in majority-inverter graphs," in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[8] L. Amaru, P.-E. Gaillardon, S. Mitra, and G. De Micheli, "New logic synthesis as nanotechnology enabler," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2168–2195, Nov 2015.

[9] "A novel basis for logic rewriting," Integrated Systems Laboratory, EPFL, Lausanne, Switzerland, Tech. Rep., 2017.

[10] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M92/41, 1992.

[11] J. Schmidt and P. Fiser, "The case for a balanced decomposition process," in *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, Aug 2009, pp. 601–604.

[12] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 7, pp. 866–876, August 2002.

[13] A. Mishchenko and M. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," in *International Workshop on Reed-Muller expansions in circuit design*, 2001, pp. 242–249.

[14] A. Mishchenko, M. Perkowski, and B. Steinbach, "An algorithm for bi-decomposition of logic functions," in *38th ACM/IEEE Design Automation Conference*, 2001, pp. 103–108.

[15] P. Fiser and J. Schmidt, "Small but nasty logic synthesis examples," in *8th. Int. Workshop on Boolean Problems (IWSBP)*, 2008, pp. 183–189.

[16] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 310–313.

[17] A. Mishchenko, S. Chatterjee, R. Jiang, and K. Brayton, Robert, "FRAIGs: A unifying representation for logic synthesis and verification," Berkeley University, Tech. Rep.

[18] A. Biere, "AIGER," http://fmv.jku.at/aiger/, 2007.

[19] A. Mishchenko, S. Chatterjee, K. Brayton, Robert, X. Wang, and T. Kam, "Technology mapping with Boolean matching, supergates and choices," ERL Technical Report, EECS Dept., UC Berkeley, Tech. Rep., 03 2005.

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

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

[22] 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 (ISCAS'85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[23] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS'89)*, May 1989, pp. 1929–1934 vol.3.

[24] Altera, "Advanced synthesis cookbook," Tech. Rep., Jul. 2011.

[25] C. Albrecht, "IWLS 2005 benchmarks," Tech. Rep., Jun. 2005.

[26] ——, "Altera's quartus university interface program (quip)," Tech. Rep., Jun. 2005.

[27] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," MCNC Technical Report, Tech. Rep., 1989.

[28] "Berkeley pla test set results," Tech. Rep., Jun. 1986.

[29] 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/

[30] V. Chickermane, J. Lee, and J. H. Pate, "A comparative study of design for testability methods using high-level and gate-level descriptions."