Towards AND/XOR Balanced Synthesis: Logic Circuits Rewriting with XOR

Ivo Háleček, Petr Fišer*, Jan Schmidt*

Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, Prague, Czech Republic

Abstract

Although contemporary logic synthesis performs well on random logic, it may produce subpar results in XOR-intensive circuits. This indicated the need of equal status of XORs and ANDs, with their respective Negation-Permutation-Negation (NPN) equivalents in logic synthesis procedures. To test the hypothesis of XOR importance, we introduced a novel logic representation with a native support of XOR gates, the XOR-AND-Inverter Graph (XAIG). As the first test, we implemented a rewriting algorithm in the logic synthesis and optimization package ABC and compared it with the standard rewriting algorithm based on the AND-Inverter Graph (AIG). The main experimental evaluation was performed in the context of a complete logic synthesis process, particularly the FPGA LUT mapping and mapping to standard cells. To eliminate algorithmic noise, input circuit descriptions were randomly modified while preserving their semantics. In the FPGA mapping, the XAIG rewriting dominated the AIG procedure in 8.6% of cases, while it was dominated in 1.6% of cases. For the standard cells mapping, the respective percentages were 3% and 1.5%. We conclude that the best rewriting is a combination of both approaches.

Keywords: Logic synthesis, rewriting, XOR, XOR-AND-Inverter Graph.

*Corresponding author

Email addresses: halecivo@fit.cvut.cz (Ivo Háleček), fiserp@fit.cvut.cz (Petr Fišer), schmidt@fit.cvut.cz (Jan Schmidt)

1. Introduction

Logic synthesis is seen as a mature process, accepted by the electronic industry. Yet there appeared example circuits, which lead to provably unacceptable results [1], [2]. Although we proved that the loss of the original, designer-conceived structure played a significant role [3], other aspects were found contributing. One of those aspects was the treatment and employment of XOR operators during synthesis. We found that to introduce XORs at the technology mapping phase is too late. If the result of logic synthesis is too large (as a consequence of neglecting XORs), technology mapping does not have enough power to rearrange the circuit. As current activities [4] show, the role and importance of XORs in logic synthesis algorithms is still an open topic.

Scalability is a primary requirement for logic synthesis. Contemporary procedures achieve scalability by iterative transformations of regions of the circuit. Such an approach is called *resynthesis*. It must assume that even the original circuit description has already some relevant structure. The number of iterations is limited by available computing time and by the ability of the procedure to maintain convergence. Layered heuristics therefore help the procedure to converge to an acceptable result within acceptable computing time.

For combinational synthesis to be scalable, the size of the region selected for a transformation must not strongly *depend* on the circuit size, and the region must be *small* enough for the intended transformation. Optimum implementations can be stored for such small regions [5], or they can be computed using exact synthesis [6], [7], [8], [9].

Various types of regions have been defined to suit particular transformations. *Windows* are defined in terms of transitive fan-in and fan-out of a selected node. They form the base of the *resubstitution* algorithm [10]. Regions called *cuts* come from FPGA technology mapping, originally intended to represent the logic implemented by a Look-Up Table (LUT) [11], [12], [13]. They started to be widely used for other purposes after efficient cut enumeration procedures had been designed [12] [5], [14]. The *rewriting* algorithm in ABC [5] is a combination

of cut enumeration with stored optimum implementations of the cut function.

The circuit representation underlying these and similar algorithms is in most cases a restricted version of a Boolean network. The restriction lies in the allowed repertory of node functions and in the fan-in of each node. For example, NAND networks used in classical library-based technology mapping [15] permit only one node function with the fan-in of two.

While Boolean networks are important for building the structure of a circuit and for selecting transformed regions, a high-performance representation of Boolean functions (without regard of the implementing structure) is sometimes required. For this purpose, Binary Decision Diagrams (BDDs) [16], [17] are widely used. The structure of the circuit is then built using another representation [18], [19], [20], [21].

In Boolean networks with a single node function only *(homogeneous networks)*, many operations, such as comparison, are of purely structural nature. Instead of Boolean reasoning, graph algorithms are used. Also, a degree of canonicity can be achieved by structural means, that is, if a node intended for insertion into the network has the same predecessors as another node, that node can be used instead. In [22], this technique is called *structural hashing*.

There is a price to these advantages. The limited repertory of node functions prevent some logic operators from being represented *directly*. They must be replaced by a sub-circuit built from permitted nodes. This may or may not correspond to reality: the actual cost of the replaced operator may be less than the total cost of the replacing nodes. An algorithm operating on such a representation is biased towards operators that can be represented directly.

Circuit representation in purely homogeneous networks tended to be large with too many equivalent representations of a single circuit. Better expressivity was achieved by introducing two *edge* types, namely, *inverting* and *non-inverting* edges. All contemporary network types use this feature.

The most widespread network of this kind is And-Inverter Graph (AIG) [23], [24], [25], [5]. The ABC system is a well-known implementation of numerous AIG-based algorithms [26]. In AIGs, an AND node with fan-in of two can,

in combination with negation on adjacent edges, express any member of the NPN class comprising the AND operator [27], [24]. The other NPN classes of two-input functions (XOR derivatives) can be represented only indirectly as subgraphs.

Another homogeneous network is Majority-Inverter Graph (MIG) [28], [6], with majority of three variables as the node function. As AND is a specialization of majority (with one input at 0), MIG can be viewed as a generalization of AIG. Yet, majority is monotonic, which forces non-monotonic functions, such as XOR, to be represented indirectly again.

The bias introduced by homogeneous networks probably caused the shift of focus towards heterogeneous networks, even at the cost of more complex algorithms. MIGs were augmented to form XOR-Majority Graphs (XMG) [4]. In the ABC system, a facility implementing Boolean networks with AND, XOR and MUX nodes together with negated edges, was established.

We can see that the circuit representation can affect synthesis algorithms. The bias towards directly represented elements has been already discussed. Another influence comes from the limited number of iterations explained above. What is a single transformation of a heterogeneous network, can require multiple steps in other networks. Hence, with some representations, the procedure can converge slower or does not find a given solution at all.

Resynthesis algorithms can be formulated on many representations, both homogeneous such as MIG [29] or heterogeneous such as XMG [4]. The difference from the original (e.g., AIG) formulation is that some operations which were structural in homogeneous networks become functional (Boolean) [30]. There are already functional operations in homogeneous networks. For example, functional hashing reuses a node already providing the required Boolean function of primary inputs rather than constructs a new node. Despite its origin in AIGs, functional hashing can be obviously used with any Boolean network.

In the above described networks, XORs are rarely represented directly, which could suggest certain algorithmic bias against them. To judge the role of XORs, we need a representation where ANDs and XORs would be equally ‘first class

citizens', with balanced roles. The structure must be simple enough to permit adaptation of most base ABC algorithms, and as close as possible to AIGs for fair comparison (which unfortunately, excludes XMGs). Also, the difficulties caused by making the network heterogeneous should be kept small.

We presented such structure, called XOR-AND-Inverter Graphs (XAIGs) in [31] and [32]. We based its experimental implementation on existing heterogeneous network in ABC, that is, on the GIA Manager facility [30].

To obtain first partial answers to the question of XOR importance, we modified one of ABC algorithms to use XAIGs. We have chosen *rewriting*, because the algorithm [5] is based on functional rather than structural matching, and hence allows easy generalizations. We evaluated XAIG rewriting against AIG rewriting experimentally, while screening the experiments from algorithmic noise. We found that XAIGs can indeed bring better results, but that the best approach is to combine both procedures.

Let us note that the evaluation of XOR importance was the leading motive rather than achieving superior results. Furthermore, the modification of the rewriting algorithm is a pilot experiment to show whether other algorithms shall also be adapted.

Even though both XMGs and ABC GIA AND-XOR-MUX-Inverter graphs are generalizations of XAIGs, they are not suitable for our purposes; the question we seek an answer for, is whether treating XORs and ANDs in a balanced way will improve the performance of logic synthesis. Therefore, the set of operators must be restricted to AND and XOR nodes only. Next, MIGs (and thus also XMGs) could be inefficient for "conventional" technology mapping [33], which could disturb the experimental comparison.

This paper is a continuation of [31] with more relevant evaluation experiments and extended discussion. The paper is organized as follows: after the Introduction and some preliminaries in Section 2, the proposed XAIG structure is described in Section 3, with implementation issues presented in Section 5. The newly introduced rewriting algorithm based on the XAIG structure is presented in Section 4. Section 6 contains experimental results. Section 7 discusses

the benefits and difficulties of the XAIG approach and Section 8 concludes the paper.

2. Preliminaries

2.1. And-Inverter Graphs

And-Inverter Graphs (AIGs) [23], [24], [5], 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.

In such a formalism, even simple Boolean functions can have many representations. To achieve some degree of canonicity, the ABC system enforces several rules for AIGs.

AIGs are constructed from primary inputs to primary outputs, assigning to each node a unique index in increasing order. This ensures parent nodes to have higher index than their children. The node with a lower index is always the left child of its parent. Due to this rule, nodes with the same pair of predecessors are structurally identical and can be recognized.

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"*) [23]. When used systematically, it ensures that no two subgraphs will have the same structure, but that the single occurrence will be reused instead.

Structural hashing still does not discover *functionally* equivalent subgraphs with different structures. ABC provides functionally reduced AIGs, FRAIGs [34]. If this approach is used in addition to structural hashing, also functional hashing of small subgraphs is performed, ensuring that there will be no functionally equivalent subgraphs in the AIG.

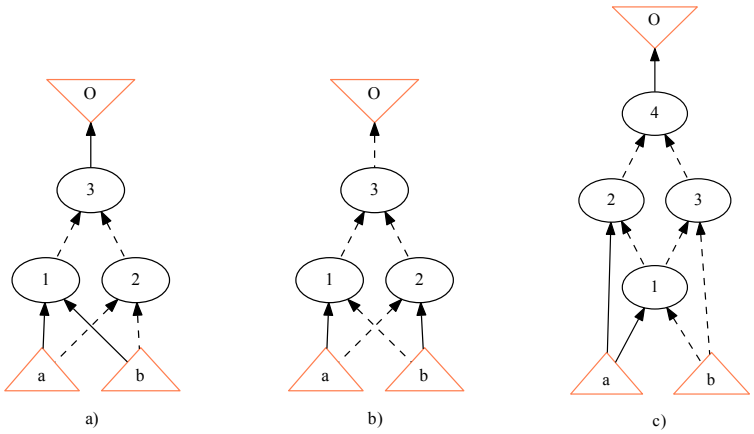

Figure 1: XOR structures in AIG.

2.2. Representation of XOR Gates in AIG

A two-input XOR gate can be represented in AIG by several structurally different ways. The minimum XOR implementation consists of three AND gates, and there are two such implementations, as shown in Figure 1a, b. Even though it is possible to construct a single XOR gate using more AND nodes, as shown in Figures 1c, such redundant structures will not be assumed here.

2.3. The Cut Enumeration Procedure

Most of scalable logic optimization algorithms work iteratively with some small subsets of the network [5], [22], [27], [10], [6], [29], [35]. For this purpose, a *cut* of a network node has been introduced [11], [12], [13], [5], [14].

A cut of a root 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.

Circuit-wise, the leaves lie on the boundary of a sub-circuit, which is able to derive the logic function of the root node solely from the values of the leaf signals.

A *cone* of a cut node N is a transitive fan-out of the cut nodes, up to the node N. In other words, the cone of the cut is the subgraph induced by the cut [12].

For the purpose of the rewriting algorithm presented in Section 4.1, maximum K-feasible cuts of all nodes must be enumerated [5]. As described in Listing 1, for a node N, the set of cuts C is created recursively, by merging cut sets of both node N children, and the trivial cut consisting of the node N itself:

$$C = merge(child1CutSet(C), child2CutSet(C)) \cup \{N\}$$

Two cut sets are merged by creating a Cartesian product of the two sets, resulting the set of unions of all pairs from both sets. The feasibility is checked for each cut, so it does not exceed K:

$$merge(C_1, C_2) = \{c_1 \cup c_2 | c_1 \in C_1, c_2 \in C_2, |c_1 \cup c_2| \leq k\}$$

Cut set of a primary input is defined as a cut consisting of the primary input itself.

A 3-feasible cut $\{6, 7, 11\}$ of the node 12 can be seen in Figure 2. This cut has been created by merging the cut $\{6, 7\}$ of node 8 and a trivial cut $\{11\}$ of node 11.

3. The XAIG Structure and its Properties

XAIG is a directed acyclic graph, where nodes are two-input ANDs or XORs, while edges can be inverted. As seen in Figure 3, XOR is described by 3 AND nodes in AIG, which can make it more difficult for algorithms to utilize it. In XAIGs, XOR is represented as a single node.

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.

The AIG techniques for partial canonicity (structural hashing) apply to AIGs as well. Functional reduction (see Subsection 2.1) can be applied to any Boolean

Listing 1: Cut enumeration algorithm

```
void NetworkKFeasibleCuts (Graph g, int k) {
  for each primary output 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) ∪ {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 ∪ Cut2| <= k) {
        Result = Result ∪ {Cut1 ∪ Cut2};
      }
    }
  }
  return Result;
}
```

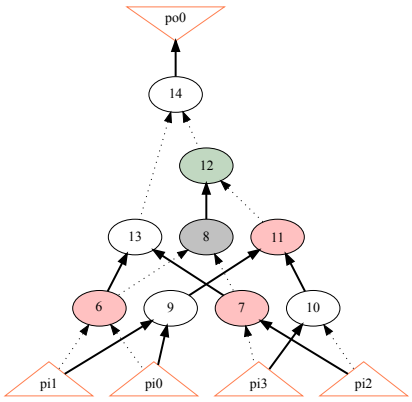

Figure 2: Cut example.

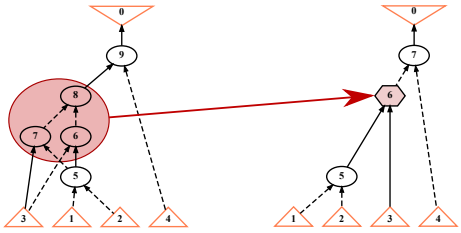

Figure 3: Logic function $F = \overline{\overline{x_1 + x_2} \oplus x_3} \cdot \overline{x_4}$ in AIG and XAIG. Circle nodes represent AND nodes, hexagon an XOR node. Dashed edges are inverted.

network. Any node can be replaced by another node providing the same function with respect to primary inputs, without affecting the function of primary outputs. Structural hashing is only a special case, characterized by identical predecessors and, in the case of heterogeneous Boolean network, by a given node function.

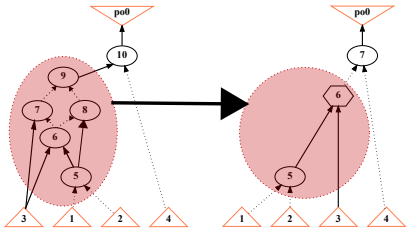

Figure 4: XAIG based rewriting example.

4. XAIG-Based Rewriting Algorithm

In order to make logic synthesis and optimization work more efficiently with XOR gates, probably all algorithms employed should be modified accordingly. As the first step towards the balanced synthesis [36] and to demonstrate whether synthesis actually needs to be capable of a native work with XOR gates, we introduced an XAIG rewriting algorithm based on AIG rewriting presented in [5].

Rewriting is a technique of replacing AIG (XAIG, in our case) subgraphs with K leaves (K-feasible cut cones [14]) by smaller, functionally equivalent precomputed structures. This algorithm can also remove functionally equivalent subgraphs from the AIG, unlike the structural hashing can. An example of a XAIG subgraph replacement can be seen in Figure 4.

4.1. 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 [14], described in Subsection 2.3 and Listing 1 (the `NetworkKFeasibleCuts` function). 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 a 16-bit integer value (for 4-feasible cuts), which is stored in a precomputed table for

Listing 2: Rewriting over XAIG network.

```
Rewrite (network XAIG, hash table PrecomputedStructures, bool
    UseZeroCost)
{
  for each node N in topological order {
    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 implementations
      bestCircuits[] = HashTableLookup(PrecomputedStructures, F);
      for each bestCircuit {
        // get XAIG with cut replaced by best circuit
        rwrXAIG = ReplaceCutByBestCircuit(XAIG, bestCircuit, cut);
        Gain = NetworkCost(XAIG) - NetworkCost(rwrXAIG);
        // keep track of best possible rewriting for the current
            node
        if (Gain > 0 || (Gain = 0 && UseZeroCost)) {
          if (bestXAIG = NULL || BestGain < Gain) {
            bestXAIG = rwrXAIG; BestGain = Gain;
          }
        }
      }
    }
    if (bestXAIG != NULL) {
        return Rewriting(bestXAIG, PrecomputedStructures,
            UseZeroCost);
    }
    else {
        continue;
    }
    return XAIG;
  }
}
```

each possible function, so are the permutations of inputs and negations of inputs and outputs needed for this conversion. Conversion to the canonical form is done by the same conversion table already available in ABC for the original AIG-based rewriting. For every truth table in a canonical form, there are one or more precomputed "best circuits" (see Subsection 4.3).

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 the output to one of 222 NPN equivalent classes [27], [5]. 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 the original AIG-based rewriting algorithm.

The node cut cone is then tried for replacement by each precomputed "best circuit". After each temporary replacement, the total network cost is calculated. If this cost is better than the cost of the original network, the replacement is made permanent.

The cost is computed 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 the expected target technology. For example, for a gate-level library targeted, the (relative) cost of 2 for AND and 5 for XOR node can be set, to reflect different sizes of the gates. When FPGA (LUT) mapping is targeted, the same cost for both node types can be set.

The *UseZeroCost* parameter in Listing 1 has been introduced for compatibility with the original ABC rewriting algorithm. When it is set to *true*, even replacements yielding zero cost improvement are accepted.

The rewriting procedure can not only reduce the number of nodes for a particular cut, also new sharing can be found in the whole network, by structural hashing. "Dissolving" of XORs is performed to find even more structural sharing, see Subsection 4.2.

4.2. XOR Transformations

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 the two 3-AND structures shown in Figures 1a, b. By this dissolving, structural sharing of XOR internal AND nodes with the rest of the network can be found. Therefore, the algorithm performs three replacement alternatives in each rewriting step (no dissolving, the structure from Figure 1a., and the structure from Figure 1b). These alternatives are compared and the one yielding the lowest total network cost is used. If such a sharing is found and it is found to reduce the total network cost, the dissolution is made permanent. Otherwise, the XOR is collapsed back to a single node.

Apart from the basic dissolving, a duplication technique can be used in addition. When the XOR cost is less or equal to x-times the cost of the AND node, x inner nodes of this XOR may be *duplicated* without negatively affecting the total cost. In particular, one or both inner AND nodes of the XOR function may be duplicated to preserve inputs for nodes outside the cut.

An example of XOR dissolving and duplication can be seen in Figure 5. An 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 an XOR node and the inner AND node (*3*) must be duplicated to preserve the input to the node *4* (network b). Another option is to preserve the XOR representation by 3 ANDs, so the node *3* will not be duplicated. However, if the XOR cost is equal to the AND cost, this alternative yields higher cost.

For experimental purposes, the XOR dissolving can be controlled by a parameter. As it was documented in [32], enabling this option produces better results in most of cases. Therefore, this option was enabled in all experiments.

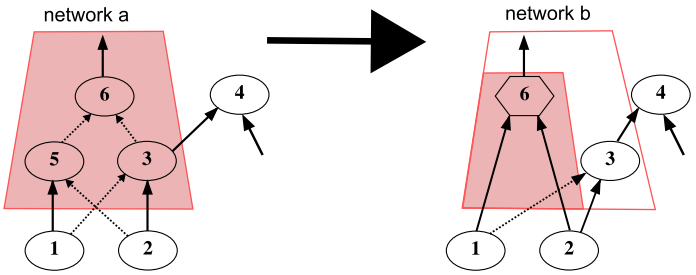

Figure 5: Duplication of AND nodes after cut replacement.

4.3. Replacement structures generation

For each of the 222 NPN equivalent classes of cut functions mentioned above, all optimal structures were pre-generated using exact synthesis [9]. Basically, the problem of finding the optimum representation of a K-input function (the Minimum circuit problem [37]) was transformed to the Satisfiability problem (SAT) and solved by MiniSAT [38]. In order to assume different XOR gates costs, Pseudo-Boolean optimization (PBO) was employed instead of SAT and solved by MiniSAT+ [39]. Enumeration of all solutions was used to generate all cost-optimal solutions. For details see [9].

For the sake of the rewriting process, the precomputed replacement structures should be "optimal". In the original AIG rewriting [5], the optimality criterion was purely the number of AND nodes. Typically, there exist more than one such "optimal" implementations of a function. From the rewriting point of view, replacing an AIG subgraph with an arbitrary minimum graph does not guarantee optimality at all, because of structural sharing (see Section 2). Therefore, multiple structures (circuits) are precomputed, "hard-wired" in the algorithm, tried for replacement, and the most efficient one is taken, as given in Subsection 4.1.

Several ways to generate these structures have been proposed in the past. In [5], the optimum replacement structures were generated by an undocumented branch&bound technique. A limited number of such structures was implemented in ABC, based on experience. In [6], [7], and [8], single optimum replacement

structures are generated on-line, by solving an SMT problem. In [29], multiple replacement structures are generated by SMT. In [35], single replacements were generated by SAT, on-line as well.

The optimality of replacement structures can be judged by numerous aspects. In [35], the cut inputs arrival times are taken into account, to minimize the delay. When more complex (i.e., more costly) nodes, like XORs, are to be present in the structure, the total area becomes of importance. Therefore, we have implemented a novel SAT and PBO-based procedure assuming different AND and XOR costs (see 4.3). Description of this method is out of the scope of this paper, for details see [9].

Because XOR dissolving is performed in the rewriting algorithm (see Subsection 4.2), structures with such dissolved XORs are not generated; all AND sub-structures describing an XOR gate are *collapsed* to a single gate. This significantly reduces the number of produced structures; since there are two ways of each XOR dissolving, the number of all possible dissolved structures grows exponentially with the number of XORs ($2^{\#XORs}$ circuits). However, by applying on-line dissolving, one structure with single-gate XORs suffices, and moreover it is processed in linear time, as described in Subsection 4.2.

The statistic on all 222 NPN-equivalent 4-input replacement circuits generated is shown in Table 1. Here the maximum and average numbers of gates (total and XORs) and levels are given, for different AND:XOR cost ratios and the case where only AND gates were allowed ("No XORs"). The maximum, average, and total counts of replacement circuits generated for each XOR cost option are shown in the last three columns. Note that the XOR structures collapsing was also applied to the "No XORs" case, producing XOR gates.

Since XORs recognized by our algorithm consist of 3 AND nodes, the recognized XORs in the "No XORs" structures consist of exactly 3 AND nodes. As a consequence, the "No XORs" structures fully correspond to the case of the AND:XOR cost ratio 1:3. Thus, these two replacements sets are equal.

In order to prove this, we have generated the replacement structures for both the "No XORs" and AND:XOR cost ratio 1:3 variants. The results were equal.

Table 1: The statistic on all generated 222 NPN-equivalent 4-input replacement circuits.

AND:XOR	**Gates**		**XORs**		**Levels**		**Count**		
cost	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Total
1:1	7	6.60	5	2.78	7	4.67	7 401	144.86	32 160
1:2	8	6.38	3	1.41	7	4.41	2 436	42.58	9 453
2:5	9	7.18	3	0.32	7	4.51	762	30.92	6 864
1:3 = No XORs	10	8.02	3	0.23	8	4.87	3056	95.45	21 190

Therefore, there is no reason for differenciating them.

We can see that with the increasing XOR cost, the total number of gates increases, the number of XOR gates decreases and the number of levels increases. Even though this conclusion is intuitive, the main purpose of Table 1 is to illustrate *how much* the XOR cost influences the result.

Approximately 70 replacement circuits on average have been generated for each function, except of the case of AND:XOR cost ratio 1. Significantly more solutions were generated here, compared to the other cost ratios. This is the first hint of the strength of the XOR gate; when AND and XOR gates can be used with the same cost, a function can usually be *optimally* implemented in many more different ways, while the solutions usually comprise XORs, even though solutions of the same cost and without XORs exist. This is illustrated in Fig. 6. The minimum implementation of the example function (negation of majority of three) consists of four gates. There are 6 structurally different optimum solutions without XOR nodes, while there are 222 optimum implementations when XORs are allowed too.

Compared to the first XAIG rewriting version presented in [31], two major improvements have been made in the replacement circuits generation: 1) using exact synthesis ensures that the generated structures are optimal in terms of the cost function mentioned above, 2) instead of using only one replacement structure per function, all optimal structures are tried for replacement, which

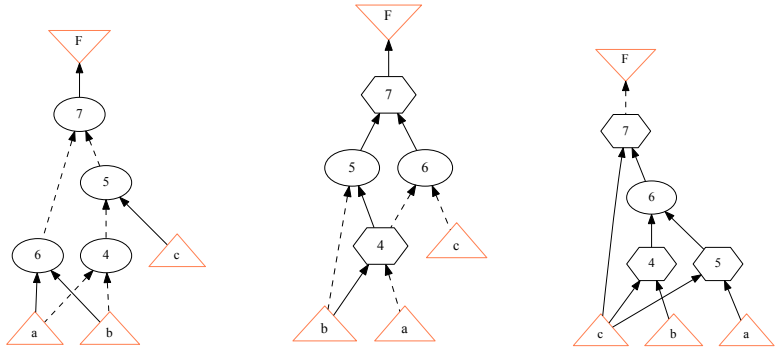

Figure 6: Three examples of optimum implementations of a function $F = \bar{a}\bar{b} + \bar{b}\bar{c} + \bar{a}\bar{c}$.

can lead to more intensive structural sharing with the rest of the graph at the cost of a higher run time.

5. Implementation of XAIGs in ABC

5.1. The ABC GIA Manager

The ABC9 package features a new manager for AIG manipulation, the GIA manager, as an alternative to the standard AIG manager [30]. The purpose of this manager is mostly experimental. Compared to the AIG manager, it is more memory-efficient and provides a faster way to search for certain structural patterns, therefore it is used mostly for the purpose of conversion between different logic representations. On the other hand, it is less effective in performing AIG modifications.

The GIA manager provides a possibility to optionally use AND-XOR-MUX-Inverter graphs, instead of standard AIGs. Here it is possible to use XOR and MUX nodes in addition to standard AND nodes. Therefore, we have 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.

5.2. 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 [25] 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 index higher than the right one, which is forbidden in the original AIG. As an example, a circuit comprised of one XOR can be described in the XAIGER format in the following way:

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

There is also a possibility to convert XAIG to the original AIG manager with the command *&put*, store it to a BLIF file [40] and later reconstruct it by loading and converting back to GIA by the already existing XOR-supporting structural hashing (ABC command *&st -m*). However, this will not only reconstruct XOR nodes already present in the XAIG, it would also convert XOR

functions represented by ANDs to XOR nodes. Therefore, this new format is necessary if we want to keep exactly the same structure and distinguish an XOR represented by a single node and an XOR represented by three AND nodes.

5.3. Recognizing XOR Gates

XOR identification was already available in the ABC9 package and could be performed 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. Setting this option to 1 disables MUX creation completely, but still allows XOR nodes creation.

This procedure identifies XOR gates represented by 3 ANDs as shown in Figures 1a, b. However, if an XOR is described by a more complex structure (i.e., Figure 1c), the *&st* command is too weak to identify it and 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.

6. Experimental Results

As a comparison of the AIG-based synthesis performance with the XAIG-based synthesis, we have run both rewriting algorithms over a set of more than 700 benchmark circuits obtained as a mix of different benchmarks: LGSynth'91 [41], IWLS'93 [42], ISCAS'85 [43], ISCAS'89 [44], ITC'99 [45], LEKO/LEKU benchmarks [1], EPFL [46] and IWLS 2005 [47] – available from [48].

6.1. Influence of the XOR Cost

There are two places where the XOR cost can be set and where it influences the result. The first point is the optimum replacement circuits generation (see Subsection 4.3). The other one is the rewriting process itself (see Subsection 4.1). Here we will investigate how combinations of these parameters influence the total number of gates and XORs after rewriting. We have processed more than 700 circuits by the *&rewrite* command with different parameters setting

Table 2: The influence of different AND:XOR cost ratios in replacement structures and during rewriting.

	&rewrite 1:1			*&rewrite* 1:2		
Generated for	Nodes	XORs	Cost	Nodes	XORs	Cost
AND:XOR cost 1:1	505 571	22 371	505 571	510 427	13 892	524 319
AND:XOR cost 1:2	509 393	17 259	509 393	511 760	13 618	525 378
AND:XOR cost 2:5	510 898	16 375	510 898	512 692	13 507	526 199
AND:XOR cost 1:3 = No XORs	510 630	16 195	510 630	512 937	13 448	526 385
	&rewrite 2:5			*&rewrite* 1:3		
Generated for	Nodes	XORs	Cost	Nodes	XORs	Cost
AND:XOR cost 1:1	513 190	11 012	529 708	517 607	8 560	534 727
AND:XOR cost 1:2	514 423	10 614	530 344	519 074	8 126	535 326
AND:XOR cost 2:5	515 242	10 651	531 219	519 775	8 241	536 257
AND:XOR cost 1:3 = No XORs	515 531	10 559	531 370	519 933	8 185	536 303

and summarized the numbers of resulting nodes and XORs. The results are shown in Table 2.

It can be observed that both parameters influence the process, even though the rewriting gate cost setting has higher significance. This can be easily understood; this parameter controls the whole rewriting flow and decides on the choice of replacement circuits to minimize the total network cost. Conversely, even though the replacement circuits contain, e.g., less XOR nodes in general (in case of high XOR costs), this fact just slightly influences the decisions.

6.2. Comparison of Rewriting Algorithms for Different XOR Costs

A comparison of the original AIG-based rewriting algorithm with the XAIG-based rewriting algorithm is presented in this subsection. Particularly, ABC commands *rewrite* and *&rewrite* were compared. The counts of resulting nodes (AND, XOR) and the numbers of circuit levels were measured.

The results are shown in Table 3. After the circuit name, nodes, XOR and levels count statistics are shown for the original AIG-based rewriting (the

Table 3: Comparison of the AIG-based rewriting (*rewrite*) and XAIG-based rewriting (*&rewrite*) with different AND:XOR cost settings.

	rewrite + &st			*&rewrite*, AND:XOR cost 1:1			*&rewrite*, AND:XOR cost 1:2			*&rewrite*, AND:XOR cost 1:3		
name	Nodes	XORs	Levels	Nodes	XORs	Levels	Nodes	XORs	Levels	Nodes	XORs	Levels
g625 [1]	9 148	2 204	24	9 000	2 500	24	9 008	1 500	24	9 383	1 125	24
s35932 [44]	7 722	694	11	6 973	901	11	7 338	886	12	7 850	630	12
b21.1 [45]	7 780	740	64	8 631	661	87	8 713	592	88	8 930	424	87
sin [46]	4 731	449	219	4 650	453	202	4 644	430	219	4 846	258	222
s38417 [44]	7 808	399	27	7 964	406	28	7 994	384	28	8 342	156	29
s38584.1 [44]	10 032	311	25	9 824	509	23	9 961	340	24	10 226	156	26
DMA [47]	21 330	254	18	21 147	349	24	21 184	310	24	21 336	205	24
c7552 [43]	1 080	294	24	1 056	326	27	1 105	295	23	1 167	179	28
adder [46]	764	255	255	637	257	256	764	255	255	1 018	1	255
b14 [45]	4 770	331	60	4 939	340	73	4 997	247	76	5 086	186	76
mem_ctrl [47]	13 156	236	35	13 030	269	36	13 040	245	36	13 221	100	36
bigkey [41]	3 510	341	11	3 396	343	12	3 397	229	11	3 848	226	12
g125 [1]	1 369	337	18	1 375	375	18	1 381	225	18	1 431	175	18
b14.1 [45]	3 885	313	57	4 000	286	61	4 048	212	69	4 096	193	69
s15850.1 [44]	2 690	204	37	2 588	246	36	2 606	203	35	2 869	52	39
Total	**606 851**	**18 772**	**12 877**	**598 802**	**25 510**	**13 035**	**603 772**	**17 195**	**13 096**	**613 691**	**10 438**	**13 281**

ABC command *rewrite*) and XAIG-based rewriting (the new ABC command *&rewrite*). Results for only 15 circuits with the highest number of XORs recognized by *&rewrite* with the 1:1 AND:XOR cost ratio are shown, while the sums over all circuits (more than 700) are shown in the last table row ("Total"). The AIG-based rewriting results were converted to XAIGs by structural hashing (the *&st* command), in order to extract XOR gates from the AIG structure. Note that this involves just simply replacing 3 AND nodes by one XOR node, where appropriate structures were found (see Figures 1a, b). The *&rewrite* command was run with different AND:XOR cost settings, the XOR node costs in the rewriting structures were set accordingly (see the AND:XOR ratios in the column labels). The AND:XOR cost ratio 2:5 was not included in the results for the lack of space, but the results were, as expected, between the ratios 1:2 and 1:3. XORs were always tried for dissolving (see Subsection 4.2).

We can see that when higher XOR nodes costs are assumed, the algorithm prefers AND nodes over XORs, and the total number of nodes naturally increases, ending up even in a situation where the number of nodes was increased w.r.t. the original rewriting, see the columns corresponding to the 1:3 ratio (where no XORs were produced in the replacement structures explicitly).

The minimum total number of nodes and also maximum of XORs was ob-

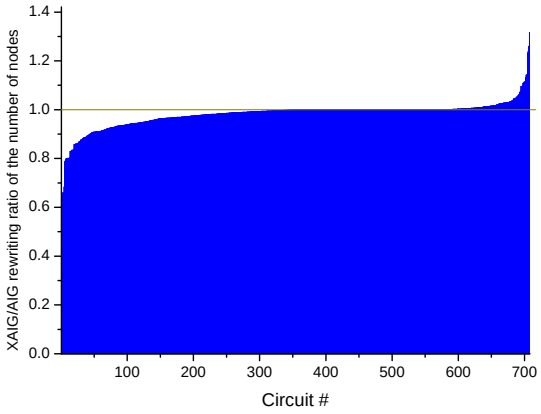

Figure 7: Comparison of XAIG-based rewriting to AIG-based rewriting by the number of nodes. Values below 1 are positive for XAIG rewriting.

tained for the XOR cost set to 1, e.g., equal to the AND cost. Particularly, the rewriting with the AND:XOR cost ratio 1:1 lead to less nodes than the converted AIG result in 351 cases (50%), while to more nodes in 124 cases (18%), out of total 708 circuits. This is also illustrated in Figure 7. The graph is composed of vertical lines, one for each circuit, with the length of the corresponding ratio, sorted in ascending order. Values below 1 indicate a better result for the XAIG rewriting (less nodes).

These results may seem to be quite obvious, however they fully expose the "strength" of the XOR operator; when XORs are allowed in addition to ANDs, the circuits can be implemented using less gates. This result confirms the theoretical reasoning on circuit complexity [49] and it justifies (and emphasizes) the XOR usefulness in synthesis.

6.3. The Overall Synthesis Process – FPGA Mapping

The previous experiment indicates that XOR introduction may benefit the synthesis process. To obtain more relevant results, both variants should be compared in the more realistic context of complete synthesis. First, technology mapping should be included. Second, the standard ABC command sequences

Table 4: Comparison of the XAIG-based rewriting to AIG-based rewriting in terms of area after FPGA mapping. Results for only 15 largest circuits are shown.

	rewrite		*&rewrite* 1:1		*&rewrite* 1:2		*&rewrite* 1:3	
name	LUTs	Levels	LUTs	Levels	LUTs	Levels	LUTs	Levels
arbiter [46]	4 053	30	4 053	30	4 053	30	4 053	30
s38417 [44]	3 008	9	2 932	9	2 945	10	2 942	9
apex2 [42]	1 696	7	1 690	7	1 679	7	1 679	7
bigkey [42]	1 695	3	1 789	3	1 898	3	1 901	3
too_large [42]	1 475	8	1 504	8	1 377	9	1 377	9
mainpla [41]	1 419	10	1 394	10	1 403	10	1 391	10
dsip [42]	1 360	3	909	3	908	3	908	3
misex3 [42]	1 358	6	1 296	7	1 285	7	1 285	7
bar [46]	1 349	6	1 349	6	1 349	6	1 349	6
des [42]	1 347	6	1 289	7	1 349	6	1 331	6
xparc [41]	1 316	11	1 319	11	1 319	11	1 330	11
spi [47]	1 252	10	1 237	10	1 246	10	1 254	10
des [41]	1 249	6	1 219	7	1 284	6	1 252	6
wb_dma [47]	1 246	8	1 230	11	1 231	11	1 233	11
g216 [1]	1 198	6	1 174	6	1 186	6	1 212	7
Total	**126 258**	**3 619**	**124 066**	**3 700**	**124 789**	**3 673**	**125 016**	**3 676**

are iterative, with more than 10 iterations of basic commands such as *rewrite* recommended in ABC manual [26]. We have compared both algorithms by the number of LUTs and levels after FPGA (4-LUT) mapping. The sequence of commands used for the AIG-based process was *rewrite; balance; if; mfs*. In order to obtain solutions that most likely converged to a local optimum, the sequence was iterated 20-times. The same sequence of commands was used for the XAIG-based process, except of using *&rewrite* instead of *rewrite*.

The results for different XOR costs are shown in Table 4. Only 15 largest circuits are shown there, with summary values for all circuits shown in the last row. We can see that the XOR cost equal to 1 produced the best results, in terms of the area (LUTs count). Particularly, better results than the AIG rewriting were obtained in 290 cases (43%), worse results in 146 cases (21%), out of 682.

6.4. Algorithmic Noise

The differences shown in Table 3 and Figure 7 are comparable with variation caused by algorithmic noise [50], [51]. From previous work, we know that the implementations of the tested algorithms are sensitive to declaration order in the input description. This ordering does not have any meaning from the perspective of the described circuit. In the following experiment, we neutralized this source of variance by averaging at least 40 runs (substantially more for smaller circuits) with randomly permuted declarations of inputs and outputs.

We found 53 cases where the XAIG-based algorithm dominated the AIG-based (that is, was systematically better than the other one). On the other hand, in 10 cases the AIG-based algorithm dominated. In all other cases, there were permutations where the XAIG-based algorithm was better, and others where it was not. The performance ratios shown in Fig. 8 were computed from average values and should not be affected by algorithmic noise. In this graph, the *&rewrite*-based synthesis exhibited a better area compared to the *rewrite*-based synthesis for 44% of the circuits and the opposite for 15% of the circuits. For the rest of circuits, the average areas were equal (the ratio was 1). In the corresponding results shown in Table 4 (the column "&rewrite 1:1", where the algorithmic noise was not suppressed, the respective percentages were 43% and 21%. Such similar values indicate, that the effect of the algorithmic noise has been heavily suppressed by the number of circuits exercised, and the results presented in the previous tables and graphs are credible enough.

The cases where either algorithm dominated exhibited very little or no algorithmic noise and the separation of the algorithms was clear. This further confirms that those values in Table 4 that are important for our conclusions are reliable even without noise elimination. Two cases, where dominance either exists or does not, but yet average values are significantly different, are shown in Figure 9.

From these experiments it is apparent that sometimes a better solution was found, sometimes worse, and here this could not be attributed to the algorithmic noise. Thus, generally speaking, XAIG-based rewriting may bring benefits for

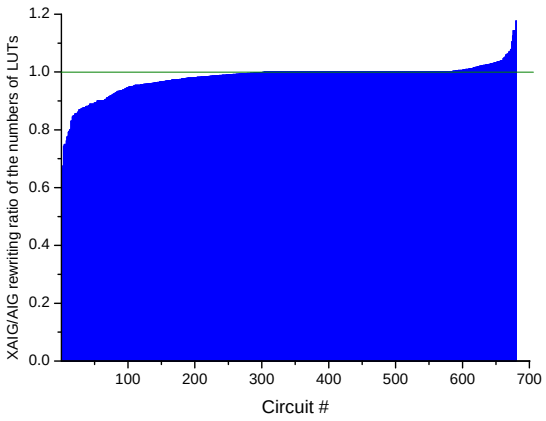

Figure 8: Comparison of XAIG-based rewriting to AIG-based rewriting in terms of area after LUT mapping. Algorithmic noise eliminated. Values below 1 indicate a better result for XAIG-based rewriting.

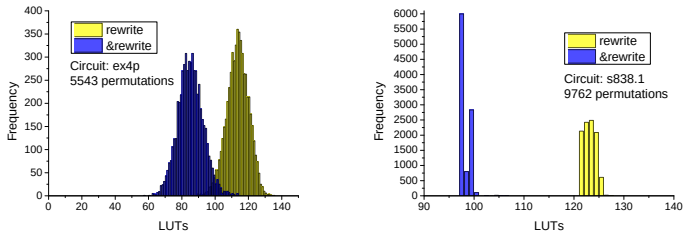

Figure 9: Histograms of result quality frequency for both algorithms and the *ex4p* [41] and *s838.1* [44] circuits.

some circuits, while for some circuits it does not help.

6.5. A Combined Synthesis Procedure

When keeping this in mind, we can suggest a general synthesis procedure, that always produces equal of better results than the original AIG rewriting based one: to run both synthesis procedures simultaneously (e.g., by employing two CPU cores) and pick the better result. However, we will show that even that is not necessary; half of the above mentioned iterations is usually sufficient to obtain better results in most of cases. This can be explained by the fact, that the iterative process (*rewrite*-based or *&rewrite*-based) quickly gets stuck in a local optimum and does not further improve much with later iterations.

This is documented in Figure 10. Here results of the *rewrite*-based process run iteratively 40-times are compared to 20 iterations of both processes (*rewrite*-based and *&rewrite*-based) with the better result taken (a choice is made). Thus, the total run times of both complete processes were approximately equal. Since the 1:1 AND:XOR cost ratio setting led to best results in the previous experiment, only this option was used here. The initial theory was confirmed – the combined process gave better results in most of cases. There were only 10 circuits for which the combined process gave slightly worse results.

6.6. Standard Cells Mapping

For completeness, we have performed a mapping into the standard MCNC gate library [52], similarly to the LUT mapping in Subsection 6.3. Particularly, the script *rewrite; balance; map; mfs* iterated 20-times was used for the original AIG-based rewriting, *&rewrite* instead of *rewrite* was used for the XAIG-based synthesis. This technology library comprises simple gates like AND, OR, NAND, NOR, XOR, XNOR, and also complex gates (AOI, OAI), with their sizes (area) and delays provided.

Since the 2-input NAND (NOR) gates cost is 2 and the XOR (XNOR) cost is 5 in this library, the AND:XOR cost 2:5 setting seems to be natural to use

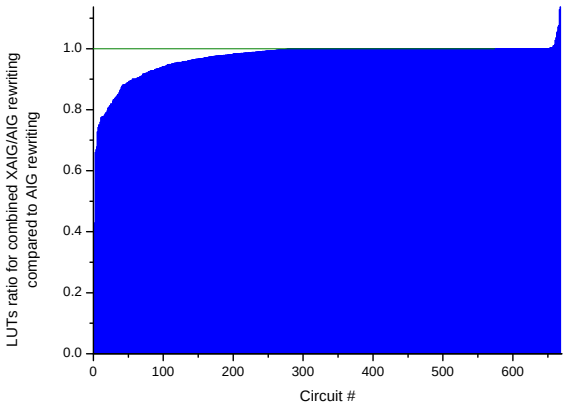

Figure 10: Comparison of the combined XAIG/AIG-based synthesis with AIG based synthesis.

for experiments. Apart from this, we have also tried out the 1:1 ratio, just for comparison.

The total areas and delays were measured (by ABC) and compared for the AIG- and XAIG-based synthesis processes, for AND:XOR cost ratios 1:1 and 2:5. The results for the 15 largest circuits are shown in Table 5, with the summary values over all processed 405 circuits shown in the last table row. The algorithmic noise (Subsection 6.4) has been eliminated in the experiment, by averaging results from at least 40 runs (substantially more for smaller circuits) with randomly permuted declarations of inputs and outputs. Therefore, Table 5 contains average values.

One may (incorrectly) observe, that there are only very small size and delay differences bvetween the AIG-based and XAIG-based rewriting, slightly benefiting the XAIG-based rewriting in area. Particularly, the areas and delays differ by less than 1% on average, for both AND:XOR cost ratios. The 2:5 AND:XOR cost ratio surprisingly gives worse results than the 1:1 one.

But still, XAIG-based rewriting (1:1 ratio) produced better results in 254 cases (63%), the AIG-based rewriting in 133 cases (33%). In the sense of algorithmic noise (Subsection 6.4), the XAIG-based rewriting dominated the

Table 5: Comparison of the XAIG-based rewriting to AIG-based rewriting in terms of area and delay after standard cells mapping. Results for only 15 largest circuits are shown.

	rewrite		*&rewrite* 1:1		*&rewrite* 2:5	
name	Area	Delay	Area	Delay	Area	Delay
arbiter [46]	18 815.5	70.7	18 815	70.7	18 814	70.7
des_area [47]	9 277.69	22.23	9 058	22.2	8 717	23.3
bigkey [42]	8 737.77	7.63	8 383.3	7.49	7 836.5	9.5
des [41]	7 778.8	13.63	7 319.11	13.72	7 312.5	13.95
spi [47]	6 121.69	22.5	6 091.22	23.01	6 144.5	24.67
wb_dma [47]	6 010.85	16.53	6 000.86	17.27	6 047	16.3
bar [46]	5 737.75	10.99	5 765.54	11.02	5 744.5	11.2
misex3 [42]	5 574.63	11.94	5 495.19	12.13	5 481.11	12.24
apex2 [42]	5 532.84	14.01	5 325.7	13.93	5 267.55	13.98
s15850 [44]	5 387.19	30.32	5 388.84	29.84	5 498	30.12
xparc [41]	5 310.02	25.98	5 255.62	25.45	5 304.5	25.05
dsip [41]	5 290.7	7.74	5 119.81	7.72	6 050.18	7.58
s15850.1 [41]	5 269.88	30.3	5 282.5	29.77	5 379.57	30.14
too_large [42]	5 219.68	14.8	5 113.23	15.09	5 085.35	15.2
prom2 [41]	5 007.68	13.81	5 059.58	13.75	4 889	13.6
Total	**500 626.83**	**5 992.48**	**496 361.99**	**6 015.85**	**498 254.34**	**6 213.72**

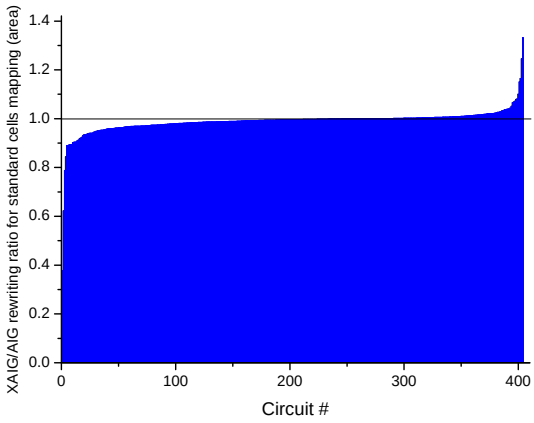

Figure 11: Comparison of XAIG-based rewriting to AIG-based rewriting in terms of area after mapping to MCNC standard cells. Algorithmic noise eliminated. Values below 1 indicate a better result for XAIG-based rewriting.

AIG-based one in 12 cases (3%), in 6 cases (1.5%) the AIG-based rewriting dominated.

The performance ratios of XAIG/AIG-based rewriting for area is shown in Fig. 11. The AND:XOR cost ratio was set to 1:1.

Similarly to the FPGA LUT mapping, sometimes a better solution was found, sometimes worse. The same conclusion can be made here as well: XAIG-based rewriting may bring benefits for some circuits, while for some circuits it does not help. In order to reach superior results in general, a combined synthesis process, as presented in Subsection 6.5, can be used.

6.7. Influence of Using Multiple Replacement Structures per Function

Finally, we will illustrate the importance of using multiple replacement circuits in the rewriting process. The results presented in [31] were inferior to the original ABC rewriting in terms of total gates counts and also in the area after the LUT mapping. This was because of using just a single replacement circuit for each function in *&rewrite*, which was not the case of the original ABC *rewrite*. Thus, we will present a comparison of XAIG rewriting processes with

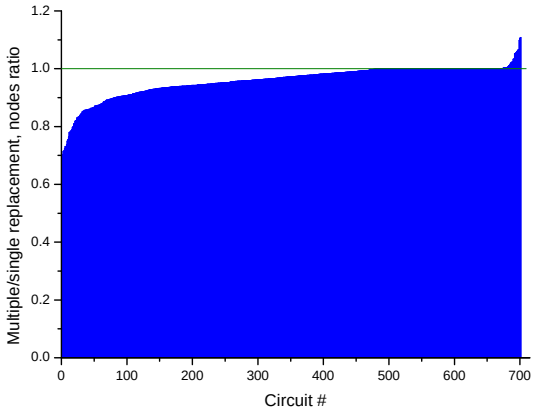

Figure 12: Comparison of XAIG-based rewriting using single and multiple replacement circuits, in terms of the number of nodes. Values below 1 indicate the benefit for multiple replacements.

and without using multiple replacements. Let us remind that 70 replacement circuits were used for each function on average.

The histogram of results is shown in Fig. 12, for *&rewrite* with XOR nodes cost set to 1. We can see that multiple replacements almost always resulted in better area. There were only 7 circuits, for which one replacement gave slightly better results.

7. Discussion

7.1. Compared procedures and replacements

To answer the question of XOR importance experimentally, two procedures should be compared, one with XORs and one without, but identical otherwise. This represents the comparison A in Figure 13. There are other factors, however, which affect the comparison.

The first one is the number of replacements. In [31], we used a single replacement for each NPN class, yielding the comparison B in Figure 13. The results were negatively affected, so that the comparison could not be fair. The

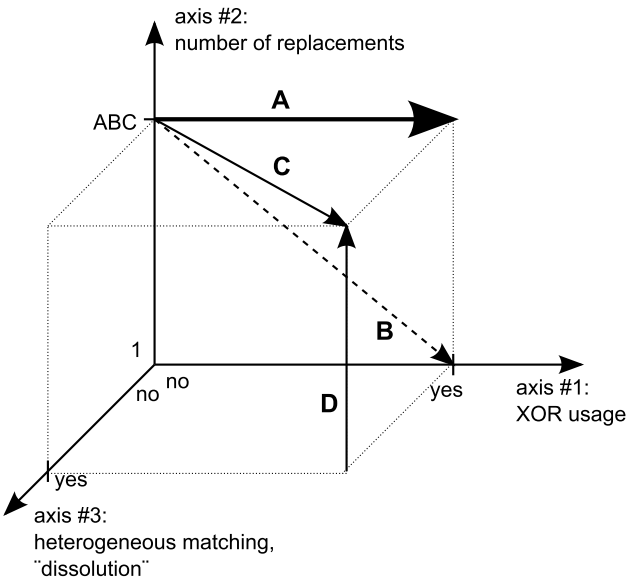

Figure 13: Factors influencing the experiments

experiment in Subsection 6.7, positioned as comparison D in Figure 13, also confirms this fact.

7.2. Replacement structures

ABC rewriting uses what can be seen as a complete set of replacements from the result quality point of view [5], [30]. NPN classes and replacements within classes are reduced only to the extent avoiding quality loss. Therefore, the replacements used in the experiments can be seen as equivalent.

The experiments in Subsection 6.1 were designed to investigate if strict correspondence of costs in replacement generation and in rewriting is necessary. The most important result, however, is that the cost of the results improves with the amount of XORs in replacements. This in part confirms our conjecture. Moreover, it permits to use the replacements generated for the AND:XOR ratio 1:1 universally, even for standard cells mapping, where the XOR cost is higher than the AND cost. This phenomenon can be explained by the fact, that the mapping process may benefit from the XOR presence, no matter what the XOR cost in the target library is. Note that the generation of structures

with the AND:XOR ratio 1:1 is easier as it does not require a PBO solver (see Subsection 4.3.

7.3. XAIG properties

XAIGs are heterogeneous networks, having more than one node type. In contrast, AIGs are homogeneous. As a consequence, purely structural methods are applicable in AIGs. For example, possible reuse of a node within the replacement can be used with little effort through *structural* hashing. For XAIGs, more complicated methods have to be used. One possibility is to dissolve XORs into ANDs and thus to obtain a homogeneous structure. Another way is to build possible transformations into replacement evaluation, as suggested by Mishchenko [30].

Heterogeneous comparison is, naturally, not needed in ABC rewriting, but can be considered an organic component of XAIGs rewriting. Thus, we see that it is actually the comparison C in Figure 13 that we want to make.

7.4. An XOR as an implicit representation

AIGs are, as mentioned earlier, a logically complete system. Nodes of any newly introduced type can be therefore replaced by subgraphs with AND nodes only. In our case, there are two distinct 3-AND subgraphs replacing an XOR. This has several consequences.

The two representations can be interpreted in the sense an XOR in XAIG *implicitly represents* two different AND-based structures. This is especially important for the number of replacement structures produced, and subsequently for the rewriting run-time. Particularly, if all XOR structures were explicitly generated as replacement circuits, their number will be *exponential* with the number of XORs ($2^{\#XORs}$). However, by representing XOR structure implicitly by one node, the rewriting time complexity is *linear* with their number, as XORs are processed one-by-one, without any dependence of previously made decisions on their dissolving.

Rewriting algorithms are based on cut generation [14], which is a purely structural procedure. When "macro" XOR nodes are introduced, less cuts can be constructed and considered for replacement, leading to worse result. This can be a possible explanation of the results in Subsection 6.2. However, as Subsection 6.5 indicates, a *combined* procedure can bring the advantages of both approaches.

8. Conclusion

A novel circuit representation structure – the XOR-AND-Inverter Graph (XAIG) was proposed in this article, together with a rewriting algorithm based on this representation. The algorithm was implemented in the framework of logic synthesis and optimization tool ABC. The XAIG-based 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 and in reducing the number of nodes than the XOR-aware structural hashing, already implemented as a command *&st* in ABC.

XOR nodes in XAIG-based synthesis bring new decisions, which need to be done, for example whether to create an XOR node even at expense of adding additional AND nodes. Most importantly, an XOR gate 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. Apart from the version presented in [31], we have implemented new options, which are configurable through *&rewrite* parameters and their influence to the final network structure has been examined. A big improvement in the XAIG rewriting results has been achieved by using all optimal replacement structures for each function as well as using exact synthesis for generation of those structures.

The impact of the XAIG-based rewriting process to a complete synthesis, particularly FPGA LUT and standard cells mapping, has been studied. When compared with the standard AIG-based rewriting process, better results have been obtained in most cases. Next, an iterative process where AIG and XAIG-

based rewriting are combined has been proposed. Better results, compared to the AIG-based rewriting process run equal time, have been obtained in a vast majority of cases.

Summarized, the newly proposed XAIG-based rewriting algorithm offers a possibility of discovering new XOR structures in a network, compared to the state-of-the-art. These XORs may be utilized in further network processing algorithms. Discovery of new XORs also yields better synthesis results in a number of cases, mostly in XOR-intensive circuits, while for the rest of circuits, comparable results are obtained. A combined procedure with superior results was demonstrated. Therefore, we can conclude that efficient and balanced handling with XORs in synthesis is useful for improving synthesis results.

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.

Computational resources were provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures".

Last, but not the least, numerous thanks to Alan Mishchenko, for his valuable comments and discussions with him.

References

[1] J. Cong, K. Minkovich, Optimality study of logic synthesis for LUT-based FPGAs, in: 14th International ACM Symposium on Field-Programmable Gate Arrays, 2006, pp. 33–40.

[2] P. Fišer, J. Schmidt, Small but nasty logic synthesis examples, in: 8th. Int. Workshop on Boolean Problems (IWSBP), 2008, pp. 183–189.

[3] P. Fišer, J. Schmidt, The observed role of structure in logic synthesis examples, in: Proceedings of the International Workshop on Logic and Synthesis 2009, 2009, pp. 210–213.

[4] W. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, A novel basis for logic rewriting, Tech. rep., Integrated Systems Laboratory, EPFL, Lausanne, Switzerland (2017).

[5] K. Brayton, Robert, A. Mishchenko, 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.

[6] L. Amaru, P.-E. Gaillardon, G. De Micheli, Boolean logic optimization in majority-inverter graphs, in: 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2015, pp. 1–6. doi:10.1145/2744769.2744806.

[7] W. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, LUT mapping and optimization for Majority-Inverter Graphs, in: International Workshop on Logic and Synthesis, 2016, p. 8.

[8] M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, Exact synthesis of Majority-Inverter Graphs and its applications, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36 (11) (2017) 1842–1855.

[9] I. Háleček, P. Fišer, J. Schmidt, SAT-based generation of optimum function implementations with XOR gates, in: 20th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, 2017, pp. 163–170.

[10] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, S. Jang, Scalable don't-care-based logic optimization and resynthesis, ACM Trans. Reconfigurable Technology and Systems (TRETS) 4 (4).

[11] J. Cong, Y. Ding, FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs, IEEE Transac-

tions on Computer-Aided Design of Integrated Circuits and Systems 13 (1) (1994) 1–12.

[12] P. Pan, C.-C. Lin, A new retiming-based technology mapping algorithm for LUT-based FPGAs, in: International ACM Symposium on Field-Programmable Gate Arrays, 1998, pp. 35–42.

[13] J. Cong, C. Wu, Y. Ding, Cut ranking and pruning: enabling a general and efficient FPGA mapping solution, in: 7th ACM/SIGDA international symposium on Field programmable gate arrays, 1999, pp. 29–35.

[14] A. Mishchenko, S. Chatterjee, K. Brayton, Robert, X. Wang, T. Kam, Technology mapping with Boolean matching, supergates and choices, Tech. rep., ERL Technical Report, EECS Dept., UC Berkeley (March 2005).

[15] K. Keutzer, DAGON: Technology binding and local optimization by DAG matching, in: Design Automation, 1987. 24th Conference on, 1987, pp. 341–347. doi:10.1109/DAC.1987.203266.

[16] S. B. Akers, Binary decision diagrams, IEEE Transactions on Computers 27 (6) (1978) 509–516. doi:10.1109/TC.1978.1675141.

[17] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691. doi:10.1109/TC.1986.1676819.

[18] K. Karplus, Using if-then-else DAGs for multi-level logic minimization, in: Proc. of Advance Research in VLSI, C. Seitz Ed, MIT Press, 1989, pp. 101–118.

[19] Y.-T. Lai, M. Pedram, S. B. K. Vrudhula, BDD based decomposition of logic functions with application to FPGA synthesis, in: Proceedings of the 30th International Design Automation (DAC'93), ACM, New York, NY, USA, 1993, pp. 642–647. doi:10.1145/157485.165078.

[20] C. Yang, M. Ciesielski, BDS: a BDD-based logic optimization system, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (7) (2002) 866–876.

[21] N. Vemuri, P. Kalla, R. Tessier, BDD-based logic synthesis for LUT-based FPGAs, ACM Transactions on Design Automation of Electronic Systems 7 (4) (2001) 501–525.

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

[23] A. Kuehlmann, V. Paruthi, F. Krohm, M. Ganai, Robust Boolean reasoning for equivalence checking and functional property verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (12) (2001) 1377–1394.

[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. Biere, AIGER, `http://fmv.jku.at/aiger/` (2007).

[26] A. Mishchenko, et al., ABC: A system for sequential synthesis and verification (2012).
URL `http://www.eecs.berkeley.edu/~alanmi/abc`

[27] Z. Huang, L. Wang, Y. Nasikovskiy, A. Mishchenko, Fast Boolean matching based on NPN classification, in: International Conference on Field-Programmable Technology (FPT), 2013, pp. 310–313. `doi:10.1109/FPT.2013.6718374`.

[28] L. Amaru, P.-E. Gaillardon, G. De Micheli, Majority-inverter graph: A new paradigm for logic optimization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (5) (2015) 806–819.

[29] M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, Optimizing Majority-Inverter Graphs with functional hashing, in: Design, Automation and Test in Europe, 2016, pp. 1030–1035.

[30] A. Mishchenko, Personal communication (2017).

[31] I. Háleček, P. Fišer, J. Schmidt, Are XORs in logic synthesis really necessary?, in: Proceedings of the 2017 IEEE 20th International Symposium on Design and Diagnotics of Electronic Circuit & Systems, IEEE, Piscataway, NJ, 2017, pp. 134–139. doi:10.1109/DDECS.2017.7934583.

[32] I. Háleček, P. Fišer, J. Schmidt, On XAIG rewriting, in: International Workshop on Logic and Synthesis (IWLS), 2017, pp. 89–96.

[33] M. Soeken, Personal communication (2017).

[34] A. Mishchenko, S. Chatterjee, R. Jiang, K. Brayton, Robert, FRAIGs: A unifying representation for logic synthesis and verification, Tech. rep., UCB/ERL, EECS Dept., UC Berkeley (mar 2005).

[35] M. Soeken, G. De Micheli, A. Mishchenko, Busy man's synthesis: Combinational delay optimization with SAT, in: Design, Automation and Test in Europe, 2017, pp. 830–835.

[36] J. Schmidt, P. Fišer, The case for a balanced decomposition process, in: 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, 2009, pp. 601–604. doi:10.1109/DSD.2009.156.

[37] V. Kabanets, J.-Y. Cai, Circuit minimization problem, in: 32th Annual ACM Symposium on Theory of Computing, 2000, pp. 73–79.

[38] N. Eén, N. Sörensson, An extensible SAT-solver, Lecture notes in computer science 2919 (2003) 333–336.

[39] N. Eén, N. Sörensson, Translating pseudo-Boolean constraints into SAT, Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 1–26.

[40] University of California, Brekeley, Berkeley logic interchange format (BLIF) (2005).

[41] S. Yang, Logic synthesis and optimization benchmarks user guide: Version 3.0, Tech. rep., MCNC Technical Report (Jan 1991).

[42] K. McElvain, IWLS'93 Benchmark Set: Version 4.0, Tech. rep. (May 1993).

[43] F. Brglez, 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.

[44] F. Brglez, D. Bryan, K. Kozminski, Combinational profiles of sequential benchmark circuits, in: IEEE International Symposium on Circuits and Systems (ISCAS'89), 1989, pp. 1929–1934 vol.3. doi:10.1109/ISCAS.1989.100747.

[45] F. Corno, M. Reorda, G. Squillero, RT-level ITC'99 benchmarks and first ATPG results, Design Test of Computers, IEEE 17 (3) (2000) 44–53. doi:10.1109/54.867894.

[46] L. Amaru, The EPFL combinational benchmark suite, Tech. rep., Integrated Systems Laboratory, EPFL, Lausanne, Switzerland (Sep. 2016).

[47] C. Albrecht, IWLS 2005 benchmarks, Tech. rep. (Jun. 2005).

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

[49] I. Wegener, The Complexity of Boolean Functions, John Wiley & Sons Ltd., 1987.

[50] J. Schmidt, P. Fišer, J. Balcárek, On robustness of EDA tools, in: Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2014, pp. 427–434.

[51] W. Shum, H. Anderson, Jason, Analyzing and predicting the impact of CAD algorithm noise on FPGA speed performance and power, in: International ACM Symposium on Field-Programmable Gate Arrays, 2012, pp. 107–110.

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