Structural Rewriting in XOR-Majority Graphs

Zhufei Chu[1] Mathias Soeken[2] Yinshui Xia[1] Lunyao Wang[1] Giovanni De Micheli[2]

[1]Faculty of Electrical Engineering & Computer Science, Ningbo University, Ningbo, China

[2]École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland

ABSTRACT

In this paper, we present a structural rewriting method for a recently proposed *XOR-Majority graph* (XMG), which has exclusive-OR (XOR), majority-of-three (MAJ), and inverters as primitives. XMGs are an extension of *Majority-Inverter Graphs* (MIGs). Previous work presented an axiomatic system, Ω, and its derived transformation rules for manipulation of MIGs. By additionally introducing XOR primitive, the identities of MAJ-XOR operations should be exploited to enable powerful logic rewriting in XMGs. We first proposed two MAJ-XOR identities and exploit its potential optimization opportunities during structural rewriting. Then, we discuss the rewriting rules that can be used for different operations. Finally, we also address structural XOR detection problem in MIG. The experimental results on EPFL benchmark suites show that the proposed method can optimize the size/depth product of XMGs and its mapped look-up tables (LUTs), which in turn benefits the quantum circuit synthesis that using XMG as the underlying logic representations.

CCS CONCEPTS

• **Hardware** → **Combinational synthesis**; **Circuit optimization**;

KEYWORDS

logic synthesis, logic networks, rewriting, majority logic

ACM Reference Format:

Zhufei Chu[1] Mathias Soeken[2] Yinshui Xia[1] Lunyao Wang[1] Giovanni De Micheli[2]. 2019. Structural Rewriting in XOR-Majority Graphs. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287671

1 INTRODUCTION

Multi-level logic synthesis plays an important role in automated design flow [7, 13]. It aims at finding a multi-level logic network of Boolean function in terms of better quality while considering different cost functions. Typical cost functions are the number of logic gates, logic depth, and switching activity, which in turn corresponded to better area, performance, and energy. To this end, efficient representation and optimization of Boolean functions are key features [1].

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00

https://doi.org/10.1145/3287624.3287671

With the continuous increase in logic design complexity, the logic representations have shifted from complex heterogenous one to simpler homogeneous networks, such as *And-Inverter Graphs* (AIGs) [14] and *Majority-Inverter Graphs* (MIGs) [1, 3]. These homogenous representations are simple and easy to manipulate, which enable more efficient optimization, requiring less memory and allowing better run times [10]. Logic rewriting is carried out to optimize logic representations, which is the most scalable optimization strategy and can be applied to very large functions. In terms of MIG rewriting, it can be either *functional*, e.g., by replacing small subnetworks up to four inputs with their optimum counterpart exploiting the results from exact synthesis [15], or *structural*, e.g., by applying the five transformation rules in an axiomatic system, Ω, and its derived three transformation rules, Ψ [1].

To obtain a more compact logic representations, it is reported that the introduction of *exclusive-OR* (XOR) operation can be significantly more advantageous over homogenous logic representations. Therefore, the extensions of AIG and MIG to *XOR-AIG* (XAIG) [11] and *XOR-MIG* (XMG) [10] are presented, respectively. XMGs have been applied in exact synthesis aware rewriting, pre-optimization for 6-LUT mapping [10], and synthesis of quantum networks. Especially, in commonly used cost models for quantum computing, majority-of-three (MAJ) can be implemented as the same cost of AND/OR and the cost of an XOR can be neglected [16]. Hence, XMGs are advantageous for quantum circuit synthesis. In order to support the natural manipulation of MIGs, a new Boolean algebra consisting of Ω and Ψ is proposed based exclusively on MAJ and inverter operations. However, a thorough consideration of MAJ-XOR logic expressions and their implementation in XMGs are not addressed in the literature.

The current XMG generation method in the literature is based on *functional rewriting* (FR) of AIGs [10], which is further improved by functional decomposition using MAJ [9]. Given an input network N, the approach proposed in [10] first maps the network into k-LUTs, e.g., in a size- or depth-oriented manner. Each k-LUT represents a k-variable Boolean function which is then used as input for exact synthesis. The results of exact synthesis are saved. Then the locally optimum networks are merged together to construct an optimized, functionally equivalent network N'.

In this paper, we aim to exploit *structural rewriting* (SR) method based on Ω and several MAJ-XOR identities that favor optimizations of size, depth, or inverters configuration in XMGs. Our contributions are as follows:

(1) We propose two MAJ-XOR identities and exploit its potential optimization opportunities during SR. Also, we discuss the rewriting rules that can be used for different operations in XMG. (Section 3).
(2) In contrast with FR method, we propose XOR structural detection method in MIG to build an XMG. (Section 4)
(3) Based on MAJ-XOR identities and XOR structural detection, we present an XMG size optimization algorithm. (Section 5)

Given an XMG obtained by FR as a starting point, experimental results on EPFL benchmarks reveal that the proposed SR method achieves with an average 32% reduction on XMG size/depth product, while 5% reduction on look-up tables (LUT) size/depth product. Also, considering the implementation cost of a T gate is extremely expensive in quantum circuit realization, the results on quantum synthesis show that the proposed method can optimize T gate count by 6% while using 5% more quantum bits (also called qubits or lines).

2 MAJORITY-INVERTER GRAPHS

Majority is a powerful generalization of AND/ORs, the MAJ function evaluates to true if and only if at least two variables are true. It can be expressed in disjunctive, conjunctive normal form, and exclusive-sum-of-products (ESOP) form as

$$\begin{aligned}\langle xyz\rangle =& xy \vee xz \vee yz = (x \vee y)(x \vee z)(y \vee z)\\ =& xy \oplus xz \oplus yz\end{aligned} \tag{1}$$

where '$\oplus$' is the XOR operation as

$$x \oplus y = x\bar{y} \vee \bar{x}y = (x \vee y)(\bar{x} \vee \bar{y}) \tag{2}$$

By setting one of its arguments to 0 or 1, the Boolean operation AND and OR can be obtained from MAJ, respectively.

$$\langle 0xy\rangle = x \wedge y \quad \text{and} \quad \langle 1xy\rangle = x \vee y \tag{3}$$

As a homogenous logic representation, MIGs use MAJ together with negation (inverter) as the only logic operations. Hence, the general *AND/OR/Inverter Graphs* (AOIGs) or AIGs are a special case of MIGs. MIG-based representations are extremely effective at logic rewriting. The axiomatic system for the MIG Boolean algebra, referred to as Ω (Eqn. (4)), is defined by five primitive transformation rules: commutativity ($\Omega.C$), majority ($\Omega.M$), associativity ($\Omega.A$), distributivity ($\Omega.D$), and inverter propagation ($\Omega.I$).

$$\Omega = \begin{cases}\textbf{Commutativity} - \boldsymbol{\Omega.C}\\ \langle xyz\rangle = \langle yxz\rangle = \langle zyx\rangle\\ \textbf{Majority} - \boldsymbol{\Omega.M}\\ \begin{cases}\langle xyz\rangle = x = y & if\ x = y\\ \langle xyz\rangle = z & if\ x = \bar{y}\end{cases}\\ \textbf{Associativity} - \boldsymbol{\Omega.A}\\ \langle xu\langle yuz\rangle\rangle = \langle\langle xuy\rangle uz\rangle\\ \textbf{Distributivity} - \boldsymbol{\Omega.D}\\ \langle xy\langle uvz\rangle\rangle = \langle\langle xyu\rangle\langle xyv\rangle z\rangle\\ \textbf{Inverter Propagation} - \boldsymbol{\Omega.I}\\ \overline{\langle xyz\rangle} = \langle\bar{x}\bar{y}\bar{z}\rangle\end{cases} \tag{4}$$

A strong property of MIGs and their algebraic framework is reachability. It has been proven that, by using a sequence of transformations drawn from the primitive five rules, it is possible to traverse the entire MIG representation space [1]. Rewriting strategies have been developed based on these rules which allow size reduction and significant depth reduction. The rewriting algorithm can be implemented more efficiently and more effectively when taking derived transformation rules into consideration. It turns out that the following three, referred to as Ψ (Eqn. (5)), are particularly helpful.

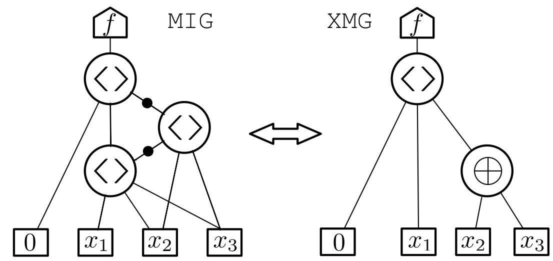

Figure 1: MIG and XMG representations of logic function $f = \langle 0, x_1, x_2 \oplus x_3 \rangle$, where the lines with solid points indicate complemented edges that require additional inverters.

$$\Psi = \begin{cases}\textbf{Relevance} - \boldsymbol{\Psi.R}\\ \langle xyz\rangle = \langle xyz_{x/\bar{y}}\rangle\\ \textbf{Complementary Associativity} - \boldsymbol{\Psi.C}\\ \langle xu\langle yu\bar{z}\rangle\rangle = \langle xu\langle yxz\rangle\rangle\\ \textbf{Substitution} - \boldsymbol{\Psi.S}\\ \langle xyz\rangle = \langle v\langle\bar{v}\langle xyz\rangle_{v/u}u\rangle\langle\bar{v}\langle xyz\rangle_{v/\bar{u}}\bar{u}\rangle\rangle\end{cases} \tag{5}$$

where $f_{x/y}$ is the expression that is obtained when replacing all occurrences of x with y in f.

3 XOR-MAJORITY GRAPHS

In this section, we first present XMGs and then exploit their associated Boolean algebra. Notable properties of XMGs are discussed.

3.1 XMG Logic Representation

The main motivation to extend MIG into XMG is the complexity of expressing the n-input parity function using MIG. The optimal MIG-based representation of 4-input parity function $f = x_1 \oplus x_2 \oplus x_3 \oplus x_4$, obtained by exact synthesis, requires nine MAJ operations as

$$\begin{aligned}f =& \langle\bar{x}_1\langle 1x_1\langle\bar{x}_3\langle\bar{x}_2x_3x_4\rangle\langle x_2x_3\bar{x}_4\rangle\rangle\rangle\\ & \langle 0x_1\langle x_3\langle x_2\bar{x}_3\bar{x}_4\rangle\langle\bar{x}_2\bar{x}_3x_4\rangle\rangle\rangle\rangle\end{aligned} \tag{6}$$

The complexity grows quickly when we add even more inputs. Efficient logic realization of arithmetic components heavily depend on 3-input parity functions. Hence, XMG that contain both MAJ and XOR can be significantly more advantageous over MIG, as shown in Fig. 1, in which each node corresponds to either MAJ (3-input) or XOR (2-input) operator and the connections between nodes can be inverted.

3.2 XMG Boolean Algebra

XOR operation has several useful properties. The basic identities are as

$$\Delta = \begin{cases}x \oplus y = 0 & if\ x = y\\ x \oplus y = 1 & if\ x = \bar{y}\\ x \oplus 0 = x\\ x \oplus 1 = \bar{x}\end{cases} \tag{7}$$

We refer to XOR Boolean algebra as Φ. XOR operations are associative ($\Phi.A$) and commutative ($\Phi.C$). Although it is not self-dual, it also allows to propagate inverters ($\Phi.I$),

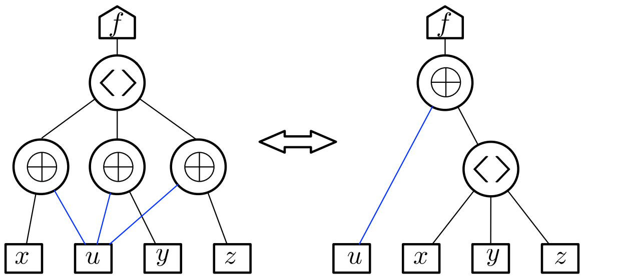

Figure 2: XMG demonstration of Theorem 1.

$$\Phi = \begin{cases} \textbf{Commutativity} - \Phi.C \\ x \oplus y = y \oplus x \\ \textbf{Associativity} - \Phi.A \\ (x \oplus y) \oplus z = x \oplus (y \oplus z) \\ \textbf{Inverter Propagation} - \Phi.I \\ \overline{x \oplus y} = \bar{x} \oplus y = x \oplus \bar{y} \\ x \oplus y = \bar{x} \oplus \bar{y} \end{cases} \tag{8}$$

Further, the XOR does not distribute over any other binary operation, but logic conjunction (and) does distribute over XOR.

$$x(y \oplus z) = xy \oplus xz \tag{9}$$

Combined MAJ and XOR operations, denoted as **MAJ-XOR**, next we exploit the Boolean algebra of MAJ-XOR operations by providing a list of identities.

Theorem 1. *XOR operation distribute over MAJ function.*

$$\langle xyz \rangle \oplus u = \langle (x \oplus u)(y \oplus u)(z \oplus u) \rangle \tag{10}$$

Proof. We expand the right hand side (RHS) expression by majority function definition in ESOP form, that is

$$\begin{aligned} & ((x \oplus u)(y \oplus u)) \\ & \oplus ((x \oplus u)(z \oplus u)) \oplus ((y \oplus u)(z \oplus u)) \\ \overset{Eqn.(9)}{=} & \; xy \oplus xu \oplus yu \oplus u \\ & \oplus xz \oplus xu \oplus zu \oplus u \oplus yz \oplus yu \oplus zu \oplus u \\ \overset{Eqn.(7)}{=} & \; xy \oplus xz \oplus yz \oplus u = \langle xyz \rangle \oplus u \end{aligned}$$

With this simplification, we conclude the proof. □

By applying Eqn. (10) from RHS to LHS (left hand side), the size (node number used in XMG) is reduced from 4 to 2, while the shared input is pushed up one level, as shown in Fig. 2. Consequently, it is possible to rewrite a MAJ node for size and depth optimization, if (i) all children are the same nodes type XOR, and (ii) there is one shared input to these XOR nodes.

The constraints to apply Eqn. (10) is tight, but there are several special cases in Eqn. (10), which may be helpful as a relaxation of constraints. We list the identities in the following item, due to the symmetry, we only list the special cases of the variable x.

(1) The shared input u equals to one of the other three inputs x, y, z. Since the XOR operation over the same variable would result in constants, it provides a special way to deal with MAJ nodes with constant inputs.

$$\begin{aligned} u = x &\Rightarrow \langle 0(y \oplus u)(z \oplus u) \rangle = u \oplus \langle uyz \rangle \\ u = \bar{x} &\Rightarrow \langle 1(y \oplus u)(z \oplus u) \rangle = u \oplus \langle \bar{u}yz \rangle \end{aligned} \tag{11}$$

(2) One of the other three inputs x, y, z equals constant 1 or 0. That provides a way to deal with MAJ nodes with two XOR-type children and one direct shared input in arbitrary polarity.

$$\begin{aligned} x = 0 &\Rightarrow \langle u(y \oplus u)(z \oplus u) \rangle = u \oplus \langle 0yz \rangle \\ x = 1 &\Rightarrow \langle \bar{u}(y \oplus u)(z \oplus u) \rangle = u \oplus \langle 1yz \rangle \end{aligned} \tag{12}$$

(3) Both above two cases happened, suppose u equals x with arbitrary polarity and y is constant, we obtained

$$\begin{aligned} u = x,\ y = 0 &\Rightarrow \langle 0u(z \oplus u) \rangle = u \oplus \langle 0uz \rangle \\ u = x,\ y = 1 &\Rightarrow \langle 0\bar{u}(z \oplus u) \rangle = u \oplus \langle 1uz \rangle \\ u = \bar{x},\ y = 0 &\Rightarrow \langle 1u(z \oplus u) \rangle = u \oplus \langle 0\bar{u}z \rangle \\ u = \bar{x},\ y = 1 &\Rightarrow \langle 1\bar{u}(z \oplus u) \rangle = u \oplus \langle 1\bar{u}z \rangle \end{aligned} \tag{13}$$

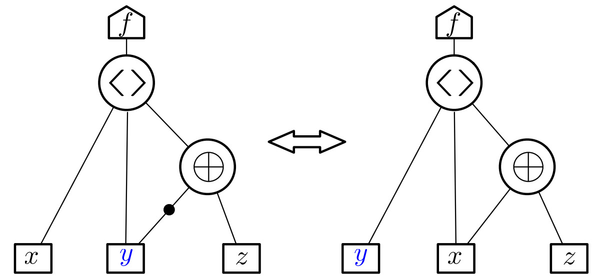

Figure 3: XMG demonstration of Theorem 2.

Theorem 2. *Complementary associativity rule exists in MAJ-XOR operations.*

$$\langle xy(\bar{y} \oplus z) \rangle = \langle xy(x \oplus z) \rangle \tag{14}$$

Proof. We expand the LHS and RHS of Eqn. (14) in ESOP form, respectively, that is

$$\begin{aligned} \text{LHS} &= (x(\bar{y} \oplus z)) \oplus (y(\bar{y} \oplus z)) \oplus xy \\ &\overset{Eqn.(9)}{=} x\bar{y} \oplus xz \oplus y\bar{y} \oplus yz \oplus xy \\ &\overset{Eqn.(7)}{=} x\bar{y} \oplus xy \oplus xz \oplus yz \\ &\overset{Eqns.(9),(7)}{=} x \oplus xz \oplus yz \end{aligned}$$

$$\begin{aligned} \text{RHS} &= (x(x \oplus z)) \oplus (y(x \oplus z)) \oplus xy \\ &\overset{Eqn.(9)}{=} x \oplus xz \oplus xy \oplus yz \oplus xy \\ &\overset{Eqn.(7)}{=} x \oplus xz \oplus yz \end{aligned}$$

Therefore, LHS = RHS, which concludes the proof. □

Theorem 2 is used to deal with reconvergent variables appearing both polarities. As shown in Fig. 3, the reconvergent variable y are pushed up one level, and the inverter is also removed. Therefore, it provides optimization opportunities in depth and number of inverters. In terms of size, if we consider special case that variable x is constant 0 and 1, we can obtain the already know identities, that are

$$\begin{aligned} x = 0 &\;\Rightarrow y(\bar{y} \oplus z) = yz \\ x = 1 &\;\Rightarrow y \vee (\bar{y} \oplus z) = y \vee \bar{z} \end{aligned} \tag{15}$$

in which the size can be also optimized.

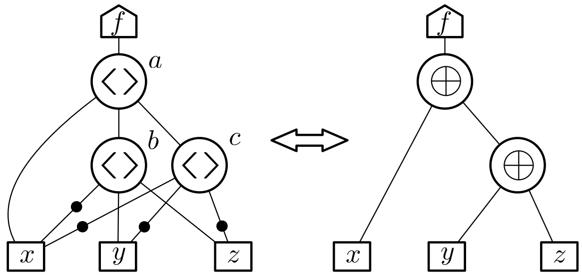

Figure 4: The MIG representation of Eqn. (16).

3.3 Rewriting Rules in XMG

Based on the rewriting rules used in MIG, and the derived Boolean algebra shown in Theorems 1 and 2, we can obtain the rewriting rules that can be used in XMGs. In terms of the MAJ nodes, the axiomatic system Ω and its derived rules Ψ are capable of utilization in XMG SR. However, compared with Ω and derived $\Psi.C$ which deal with MAJ nodes appeared at two logic levels, e.g., an MAJ parent node with one child MAJ node, the rules $\Psi.R$ and $\Psi.S$ using replacement operation to handle multi-level logic network. These operations pose additional structural constraints to require the replaced nodes and its transitive fanout nodes are not used by any other logic functions. Moreover, as the XMG is a heterogenous logic network that incorporates both MAJ and XOR operations, the implementation of replacement operation is much more complex than the counterpart in MIG. Consequently, we only consider Ω plus derived $\Psi.C$ for MAJ nodes rewriting. With regard to XOR nodes, Φ can be used, while Theorems 1 and 2 could be used to all MAJ-XOR nodes.

4 DETECTING XOR OPERATIONS IN MIG

Initial XMGs can be transformed from AIGs using the FR method proposed in [10] or functional decomposition method [9]. Due to the computation complexity of exact method and the dependency of technology mapping result, it is necessary to develop a structural detection method to find XOR operations in MIG in a separated way.

From [1], we know the depth-optimal MIG representation of 3-input parity function.

$$\langle x\langle \bar{x}yz\rangle\langle \bar{x}\bar{y}\bar{z}\rangle\rangle = x \oplus y \oplus z \tag{16}$$

By exploiting the MIG representation in Fig. 4, there are several rules in MIG to detect XOR operations. Both MAJ nodes b and c have the same logic levels, while the MAJ node a has one level larger than nodes b and c.

- x feeds the nodes b and c with the same polarity, while has the complemented polarity to fed the node a.
- y and z feed the nodes b and c with complemented polarities.

To specify one of variables from $\{x, y, z\}$ to constant input, we can obtain 2-input XOR operation. However, there is still one special case to discuss, that is variables do not appear as complemented polarities, but as a combination of one variable with the other one as constant input. For example, we replace one of variables $\{\bar{x}, \bar{y}, \bar{z}\}$ in Eqn. (16) as constant input 1 or 0, we obtain following identities.

$$\langle x\langle 1yz\rangle\langle 1\bar{y}\bar{z}\rangle\rangle = \langle 1x(y \oplus z)\rangle \qquad \langle x\langle 0yz\rangle\langle 0\bar{y}\bar{z}\rangle\rangle = \langle 0x(\overline{y \oplus z})\rangle$$
$$\langle x\langle \bar{x}yz\rangle\langle \bar{x}1\bar{z}\rangle\rangle = \langle 1y(x \oplus z)\rangle \qquad \langle x\langle \bar{x}yz\rangle\langle \bar{x}0\bar{z}\rangle\rangle = \langle 0y(\overline{x \oplus z})\rangle$$
$$\langle x\langle \bar{x}yz\rangle\langle \bar{x}\bar{y}1\rangle\rangle = \langle 1z(x \oplus y)\rangle \qquad \langle x\langle \bar{x}yz\rangle\langle \bar{x}\bar{y}0\rangle\rangle = \langle 0z(\overline{x \oplus y})\rangle$$

Input : An XMG α
Output : An optimized XMG α'

if enable_xor_detection **then**
  xor_detection(α); `// Section 4`
end
for $i = 0$; $i < effort$; i++ **do**
  elimination(α);
  reshaping(α);
  elimination(α);
end
function *elimination(α)*
  $\Omega.M_{L\to R}(\alpha)$; $\Omega.D_{R\to L}(\alpha)$; `// MAJ nodes`
  Δ; `// XOR nodes`
  Eqn. $(10)_{R\to L}(\alpha)$; `// MAJ-XOR nodes`
function *reshaping(α)*
  $\Psi.C(\alpha)$; `// MAJ nodes`
  $\Phi.A$; `// XOR nodes`
  Eqn. $(14)_{L\to R}(\alpha)$; `// MAJ-XOR nodes`

Algorithm 1: XMG size optimization

5 XMG SIZE OPTIMIZATION

The optimization on graph-based logic networks ultimately consists of its transformation from one into a different another one, with better figures of merit of size, depth, or switching activity. In the rest of this section, we present heuristic algorithms to optimize the size of an XMG using rewriting rules described above.

In MIG size optimization, the proposed method in [3] applying the majority rule ($\Omega.M$) from left to right (L→R), and distributivity rule ($\Omega.D$) from right to left (R→L) to reduce the number of MIG nodes. The size optimization procedure is a loop algorithm, which would be repeated until no improvement exists. It generally contains two sub-procedures, that is elimination and reshaping. While elimination is straightforward, the reshaping is indispensable to reconstruct the MIG to provide more opportunities for the next loop, which is like permutation of solutions used in evolutionary algorithms to escape local minima.

In terms of XMG size optimization, we can also use $\Omega.M_{L\to R}$ and $\Omega.D_{R\to L}$ for MAJ nodes elimination. For XOR nodes, from the identities presented in Δ, if the two inputs of XOR are the same with arbitrary polarities, then the number of XOR nodes can be reduced. The above strategies are used separately for MAJ and XOR nodes, respectively. With the target of MAJ-XOR nodes, the node elimination opportunity arises from the identity shown in Eqn. (10), evaluated from right to left.

To reshape the XMG, as we discussed in Section III.3.3, only $\Omega.A$ and $\Psi.C$ can be applied for MAJ nodes. Besides, the *push-up* axioms proposed in [3] are also used for MIG depth optimization. The $\Phi.A$ can be applied to XOR nodes, and the identity shown in Eqn. (14) can be used for MAJ-XOR nodes. The elimination and reshaping procedure can be iterated over a user-defined number of cycles, called effort. The XOR detection method can be adopted during the iteration or implemented at the beginning but just once. However, the experimental results show the former case may consume too much CPU time. In contrast, the latter case is more reasonable. If the XOR detection operation is enabled, then we perform the method at the beginning of the algorithm. The XMG size optimization algorithm is summarized in Algorithm 1.

Table 1: Using Structural Rewriting for XMG-Size Optimization

Benchmarks	Previous method [10]				Our Method						
	XMG size	XMG depth	LUT size	LUT depth	Time (s)	XMG size	%	XMG depth	%	LUT size	LUT depth
adder	383	129	192	64	0.03	383	–	**128**	0.78	192	64
div	41,462	4,381	13,862	1,069	7.32	**39,234**	5.37	**2,573**	41.27	**12,462**	1,080
log2	21,624	210	7,672	87	3.21	**21,579**	0.21	227	–	**7,439**	85
max	2,109	273	809	72	0.16	**2,103**	0.28	273	–	846	79
multiplier	15,816	134	5,373	64	8.54	**15,785**	0.20	135	–	**5,365**	64
sin	3,842	152	1,423	48	0.48	**3,826**	0.42	156	–	1,441	**45**
sqrt	17,357	6,046	5,644	1,082	2.50	**17,289**	0.39	**5,121**	15.30	5,990	**1,051**
square	12,277	155	3,815	61	1.30	**12,232**	0.37	157	–	3,815	61
arbiter	12,159	89	3,819	20	0.92	**10,621**	12.65	**76**	14.61	**3,752**	**19**
cavlc	715	17	139	4	0.10	**706**	1.26	21	–	139	4
ctrl	118	8	29	2	0.01	**116**	1.69	**7**	12.50	29	2
i2c	1,287	18	379	5	0.16	**1,264**	1.79	19	–	**372**	5
int2float	247	15	56	4	0.02	**245**	0.81	17	–	56	4
mem_ctrl	42,580	125	12,727	35	12.64	**42,019**	1.32	135	–	12,736	38
priority	753	245	238	31	0.13	**750**	0.40	**244**	0.41	233	31
router	211	46	91	11	0.02	212	–	46	–	97	**8**
voter	6,838	60	2,429	16	1.38	**6,737**	1.48	70	–	**2,163**	**15**
Avg.	10,575	712	3,453	157	2.29	**10,300**		**553**		**3,360**	**156**
Im.	1	1	1	1		**0.97**		**0.78**		**0.97**	**0.99**
Avg. size · depth		17,847,471		1,345,881					**12,167,283**		**1,276,777**
Im.		1		1					**0.68**		**0.95**

Avg. : Average, Im. : Improvement (new/old), –: no improvement
LUT size and depth generated by 6-LUT technology mapping using ABC command `if -K 6`

6 EXPERIMENTAL RESULTS

We evaluate the proposed XMG SR method in the following sections. All experiments have been carried out on an Intel i7-4870HQ CPU at 2.50 GHz with 16 GB of main memory.

6.1 Methodology

We implemented our approach in C++ as a command called '`xmgre`' on top of the logic synthesis framework CirKit.[1] The benchmarks considered are general combinational circuits from ISCAS [8] and EPFL benchmark suites [2]. Our SR results are verified by '`cec`' command in ABC [6] to ensure functional correctness.

6.2 Results

Evaluation on EPFL Benchmarks. The experimental results are shown in Table 1. The "Benchmarks" column lists the benchmark name. Given the results produced by '`xmglut -k 4`' in CirKit [10], our SR algorithm is applied to further optimize XMG, which is a post optimization strategy. The EPFL benchmark suites contain 20 benchmarks. We list 17 of them while the remaining 3 got exactly the same results.

In terms of XMG size, the number of nodes can be reduced by 3% on average, while the XMG depth is reduced by 22%, compared to [10]. The improvement of the results are highlighted in the table. Among 17 benchmarks, 16 of them can be optimized in size and 6 in depth. Generally, the XMG size can be reduced with an overhead of XMG depth. The exceptions are benchmarks `div`, `sqrt`, `arbiter`, `ctrl`, and `priority`, which achieves both size and depth improvement. For instance, `arbiter` achieves up to 12.65% size improvement, from 12159 to 10621 nodes, and 14.61% depth improvement, from 89 to 76 levels. The depth improvement mainly contributed by `div` and `arbiter`. They former one reduced the depth from 4381 to 2573 levels, which is equal 41.27% reduction. The latter one also got a 14.61% depth reduction.

[1]github.com/msoeken/cirkit

We set the rewriting algorithm terminated when the nodes can not be improved after at least two efforts. Therefore, the rewriting cycles are distinct for benchmarks. The CPU time listed in the table indicates our SR method averagely consumes 2.29 seconds.

We also compares the results after 6-LUT mapping. Generally, XMG size optimization advantage also carries over into LUT mapping improvements in a vast majority cases. However, optimization of the size and depth of a logic network may not essentially result in reduced LUT size and depth [12]. For example, benchmarks `max` and `mem_ctrl` can be optimized in terms of XMG size, whereas results in an increasement of LUT size. On average, our method achieves 3% reduction of LUT size and 1% reduction of LUT depth. By evaluating the size/depth product metric, our method achieves 32% reduction of XMG, while 5% reduction of LUT after technology mapping.

Evaluation on ISCAS Benchmarks. Since XOR detection method is proposed, we can also proceed SR directly on AIGs, which can be transposed into MIGs by adding constant inputs. To compare the proposed rewriting method for size optimization, we compare several rewriting method using the same ISCAS benchmarks. The results are shown in Fig. 5, where the MIG rewriting method are used as baseline. The detail commands used in Cirkit to implement these methods are shown below, while the start point is reading a circuit file in AIGER format [4].

MIG SR [3]	`mig_rewrite --metric 1`
XMG SR	`aig > mig; xmgre`
XMG FR [10]	`xmglut -k 4`
XMG FR + SR	`xmglut -k 4; xmgre`

As can be seen in Fig. 5, since we add XOR detection strategy, our SR method can achieve better results than pure MIG rewriting. While the XMG FR method generally performs better then the structural method, by adding our proposed algorithm as a post optimization step, the size of some benchmarks can be further improved.

We have witnessed some circuits optimized by FR got worse XMG size results than the original AIGs. In contrast, our SR method

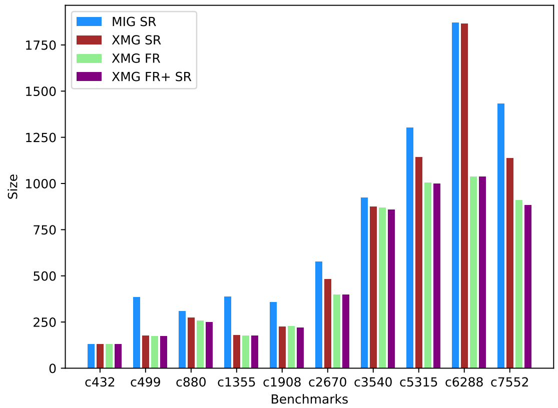

Figure 5: Comparison of different methods for XMG size optimization.

can avoid this case. This is mainly because our method do not depend on the technology mapping results. Through efficient XOR detection, we can even achieve better results than FR method. For example, the XMG size of benchmark `c1908` is 227 by FR, while 226 by SR.

Despite these circuits where FR performed less well than the original ones, FR method generally produce better results than our SR method. The main reason is technology mapping used FR already adopts intelligence of area- and depth-oriented optimization. Therefore, it makes sense to exploit "FR + SR" simultaneously to guarantee eventually better results.

6.3 Evaluation on Quantum Circuit Synthesis

XMGs have been applied in quantum circuit synthesis [16]. The basic principle is to map each gate in an XMG into a quantum network and then compose these networks [9]. As quantum computers can only implement reversible computations, auxiliary qubits are used to store intermediate results. Besides, the number of T gate is measured as the cost of a quantum network. Given the results produced in [9] over the integer reciprocal design `INTDIV(n)` for n = 16, 32, 64, and 128, we use the proposed SR to optimize XMG. The results shown in Table 2 indicate our method can further optimize the T gate count by 6% with additional 5% qubits. Since the T gate count accounts for the far most complex execution in a quantum computer [5], the optimization of T gate count has a significant impact on quantum circuit realization.

6.4 Discussions

Although the SR method can optimize XMG size, due to the derived transformation rules can only be implemented in two logic levels, the optimization results have space for further improvement. There are more optimization obstacles of heterogenous logic representation than the homogenous ones. As next steps, we aim to integrate both FR and SR to exploit more robust XMG size optimization method. Moreover, as ABC technology mapper `if` is based on AIGs instead of MIGs or XMGs, both FR and SR results may be improved if an MIG-based technology mapper is developed. The results demonstrated show the XMG quality improvements diminish after technology mapping, the mapping-aware structural rewriting is also of high interest.

Table 2: Experimental Results on Quantum Circuit Realization of Reciprocal Operation (`INTDIV`(n))

	Before rewriting [9]		After XMG rewriting		
n	qubits	T-count	qubits	T-count	CPU time (s)
16	429	14140	452	13258	0.26
32	1705	59066	1786	56028	0.46
64	6810	238182	7169	223426	2.05
128	26688	921368	28105	864276	31.52
Avg.	1	1	1.05	0.94	

7 CONCLUSIONS

In this paper, we proposed a structural rewriting method for XOR-Majority Graphs. By exploiting XOR and majority Boolean logic identities, we found the optimization opportunities during XMG structural rewriting. By evaluating EPFL and ISCAS benchmarks, the experimental results show we can further optimize XMG size/depth product by 32% and LUT size/depth product by 5% by giving an functional rewriting XMG as a starting point. The proposed method is also applied for quantum circuit synthesis, which reduce the T gate count by 6%.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for useful comments. The work was partly supported by the National Natural Science Foundation of China under Grant Nos.: 61501268, 61871242 and 61471211 and by the Swiss National Science Foundation under Grant No.: 200021-169084 MAJesty.

REFERENCES

[1] Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2014. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *DAC*. 1–6.
[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *IWLS*. 57–61.
[3] Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2016. Majority-Inverter Graph: A new paradigm for logic optimization. *IEEE Trans on Computer-aided Design of Integrated Circuits and Systems* 35, 5 (2016), 806–819.
[4] Armin Biere, Keijo Heljanko, and Siert Wieringa. 2011. *AIGER 1.9 And Beyond*. Technical Report. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria.
[5] Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A* 71, 2 (2005), 022316.
[6] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *CAV*. 24–40.
[7] Robert K Brayton, Gary D Hachtel, and Alberto L Sangiovanni-Vincentelli. 1990. Multilevel logic synthesis. *Proc. IEEE* 78, 2 (1990), 264–300.
[8] F Brglez and H Fujiwara. 1985. A neutral netlist of 10 combinational benchmark circuits and a target translator into FORTRAN. In *ISCAS*. 659–662.
[9] Z. Chu, M. Soeken, Y. Xia, and G. De Micheli. 2018. Functional decomposition using majority. In *ASP-DAC*. 676–681.
[10] Winston Haaswijk, Mathias Soeken, Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. A novel basis for logic rewriting. In *ASP-DAC*. 151–156.
[11] Ivo Háleček, Petr Fišer, and Jan Schmidt. 2017. Are XORs in logic synthesis really necessary?. In *DDECS*. 134–139.
[12] Gai Liu and Zhiru Zhang. 2017. A parallelized iterative improvement approach to area optimization for LUT-based technology mapping. In *FPGA*. 147–156.
[13] Giovanni De Micheli. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education.
[14] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC*. 532–535.
[15] Mathias Soeken, Luca Gaetano Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2017. Exact synthesis of majority-inverter graphs and its applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017), 1842–1855.
[16] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. 2017. Design automation and design space exploration for quantum computers. In *DATE*. 470–475.