Advanced Functional Decomposition Using Majority and Its Applications

Zhufei Chu, *Member, IEEE*, Mathias Soeken, *Member, IEEE*, Yinshui Xia, Lunyao Wang, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—Typical operators for the decomposition of Boolean functions in state-of-the-art algorithms are AND, exclusive-OR (XOR), and the 2-to-1 multiplexer (MUX). We propose a logic decomposition algorithm that uses the majority-of-three (MAJ) operation. Such a decomposition can extend the capabilities of current logic decompositions, but only found limited attention in the previous work. Our algorithm make use of a decomposition rule based on MAJ. Combined with disjoint-support decomposition, the algorithm can factorize XOR-majority graphs (XMGs), a recently proposed data structure which has XOR, MAJ, and inverters as only logic primitives. XMGs have been applied in various applications, including: 1) exact-synthesis-aware rewriting; 2) preoptimization for 6-input look-up table (6-LUT) mapping; and 3) synthesis of quantum circuits. An experimental evaluation shows that our algorithm leads to better XMGs compared to state-of-the-art algorithms based on XMGs, which positively affects all of these three applications. As one example, our experiments show that the proposed method achieves an average of 10% and 26% reduction on the LUTs size/depth product applied to the EPFL arithmetic and random control benchmarks after technology mapping, respectively.**

***Index Terms*—Disjoint-support decomposition, functional decomposition, logic synthesis, majority operation.**

I. Introduction

THE ADVANCEMENT of electronic design automation (EDA) tools and CMOS technologies are the main driving forces of modern digital integrated circuits (ICs). As CMOS dimensions are reaching their physical limits [1], the arrival of post-CMOS nanotechnologies poses great challenges on the innovation of EDA tools. Novel logic abstractions of new devices and synthesis techniques are indispensable to unleash the real power of the candidate nanotechnologies [2].

The building-block operations of most established computational paradigms are based on AND/OR/NOT, or NAND/NOR, or their combinations with exclusive-OR (XOR), and the 2-to-1 multiplexer (MUX). Algorithms using these operators are well developed to synthesize current digital electronics. In recent years, there has been a notable research effort into logic synthesis using the majority-of-three function $\langle xyz \rangle = xy + xz + yz$ (MAJ, [3]) and its combinations with XOR. The reasons can be attributed to the following three aspects.

1) Nanotechnologies, such as Quantum-Dot Cellular Automata [4], Spin Wave Devices [5], and Nanomagnets [6], realize MAJ as primitive building blocks.
2) In commonly used cost models for fault-tolerant quantum computing, MAJ can be implemented at the same cost as AND/OR, and the cost of an XOR gate can be neglected [7].
3) Arithmetic functions make extensive use of AND and XOR [8]. MAJ includes AND/OR, but it is more expressive than them. Therefore, it would be advantageous to enable logic synthesis methods to consider AND/OR and XOR/MAJ representations to support different circuit designs.

Logic representations that use MAJ and NOT as basic logic primitives have been recently proposed for the synthesis of Boolean logic functions [9]. The graph representation is named as majority-inverter graph (MIG) and analogously to and-inverter graph (AIG) [10]. The synthesis methods based on MIGs demonstrated superior results for both standard CMOS and emerging technologies [11]. In addition, introducing XOR as a logic primitive, the XOR-majority graph (XMG, [12]) is proposed to obtain more compact logic network. XMGs can speed-up exact synthesis since it has a high computational complexity [13]. MIGs and XMGs have been applied in various applications. Recent studies show how the combination of exact synthesis and logic rewriting led to improvements in AIG [10], [14], MIG [15], [16], and XMG size optimization [12]. However, the existing exact synthesis methods only exploit local logic networks for rather small functions, which results in optimized global logic networks for certain instances only.

The task of exact synthesis is to find an optimum logic network for the given Boolean function. Take the Boolean Satisfiability (SAT) technique as an example, exact synthesis is executed by solving sequences of SAT formulas. Using size as an optimization objective, the idea is to use an SAT solver to check whether there exists a Boolean network of r gates that realizes the given functions [17]. The algorithm starts with $r = 0$ (for constants or projection functions) and incrementally increases r until the SAT solver returns a satisfiable solution.

Manuscript received September 27, 2018; revised January 18, 2019 and April 15, 2019; accepted May 24, 2019. Date of publication June 27, 2019; date of current version July 17, 2020. This work was supported in part by NSF, China, under Grant 61871242 and Grant 61571248, in part by the Zhejiang Provincial NSF under Grant Y19F040013, in part by K. C. Wong Magna Fund in Ningbo University, under Grant H2020-ERC-2014-ADG 669354 CyberCare, and in part by the Swiss National Science Foundation under Grant 200021-169084 MAJesty. This paper was recommended by Associate Editor J. Cortadella. *(Corresponding author: Zhufei Chu.)*

Z. Chu, Y. Xia, and L. Wang are with the Faculty of Electrical Engineering and Computer Science, Ningbo University, Ningbo 315211, China (e-mail: chuzhufei@nbu.edu.cn).

M. Soeken and G. De Micheli are with the Integrated Systems Laboratory, École Polytechnique Fédéderale de Lausanne, 1015 Lausanne, Switzerland.

Digital Object Identifier 10.1109/TCAD.2019.2925392

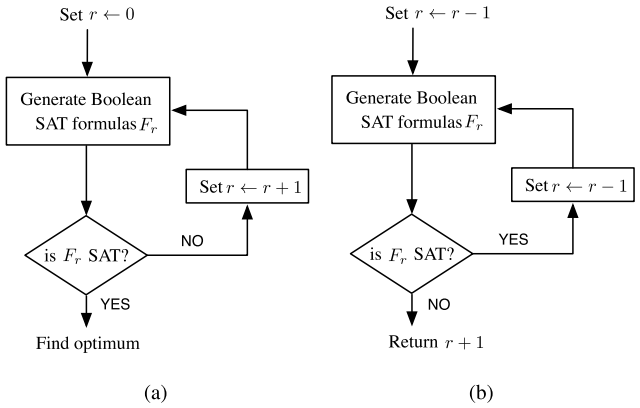

Fig. 1. Size-optimum SAT-based exact synthesis starting from (a) lower bound and (b) upper bound.

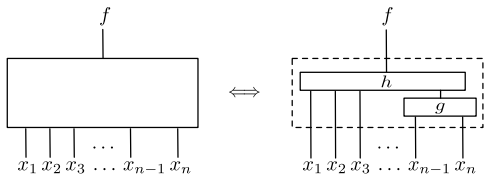

Fig. 2. Boolean functional decomposition.

This process is illustrated in Fig. 1(a). That means the search for a size-optimum network with r gates requires to solve up to $r+1$ decision problems using an SAT solver.

Boolean functional decomposition that represents a (complex) Boolean function in terms of a basis consisting of simple subfunctions (shown in Fig. 2), provides potential advantages for exact synthesis. The major drawback of exact synthesis is the potentially long runtime of SAT executions [18]. Given an n-variable Boolean function $f(x_1, \ldots, x_n)$, the runtime requirements to find an exact synthesis solution for f grow significantly for a large n. Exact synthesis can be used as a subroutine for synthesizing a large function. As shown in Fig. 2, by decomposing f into smaller subnetworks using disjoint-support decomposition (DSD) or support-reducing decomposition techniques [19], the subfunctions g and h in the decomposed network have fewer number of inputs, and exact synthesis may therefore solve them more efficiently. Further, by expanding a decomposed logic network using majority logic decomposition, it can be used as the starting point and an upper bound for exact synthesis. As shown in Fig. 1(b), instead of starting from a lower bound to find an optimum solution, the upper bound is provided as the starting point for the SAT solver to enable incremental improvement. Given as starting bound the size r of the decomposed logic network, the algorithm tries to find a solution with $r \leftarrow r-1$ gates and incrementally decreases r until the SAT solver returns an unsatisfiable solution. Then, the satisfiable solution of the last satisfiable step is returned, that is a network with $r+1$ gates.

In this paper, we extend the capability of current logic decomposition methods by additionally using MAJ. A preliminary version of this paper was demonstrated in [20]. Our contributions are as follows.

1) We make use of an MAJ decomposition which resembles the well-known Shannon decomposition: we show that under some conditions it is possible to write a function $f(x_1, \ldots, x_n)$ as $\langle zgh \rangle$ such that Boolean functions g and h do not depend on subfunction z (Section IV).
2) We propose a decomposition algorithm that combines MAJ (both from top-down and bottom-up decomposition scenarios), DSD using other ordinary operators, and Shannon decomposition to factorize an XMG from a function provided as a truth table (Section V).
3) We improve exact-synthesis-aware logic rewriting, in sense of deriving optimum or near-optimal XMGs for each look-up table (LUT) in an LUT-network (Section VI).

We conduct experiments on EPFL/ISCAS benchmark suites as well as selected Boolean functions that frequently occur in practical synthesis and technology applications. Using MAJ enables more logic decomposition opportunities. For evaluations on the Boolean functions, there is a 14% XMG size and an 8% XMG depth improvement after introducing MAJ decomposition. The experimental results over EPFL benchmark suites show that the proposed method achieves a better size/depth product of both XMGs and its mapped LUTs. Specifically, we obtained an average of 10% size/depth product reduction on XMGs and 10% on LUTs for arithmetic benchmarks part, compared with a 30% reduction on XMGs and 26% on LUTs for random control benchmarks part. Moreover, the evaluation of ISCAS benchmarks shows our method results in less LUTs, compared with using an AIG aggressive size optimization script. Finally, we also demonstrate that the proposed method is beneficial to the synthesis of quantum circuits.

The remainder of this paper is organized as follows. Section II provides a literature review of functional decomposition. In Section III, definitions on Boolean functions, logic representations, DSD, exact synthesis, and NPN classification are described. Then, in Section IV, we discuss the MAJ decomposition theoretical properties from both top-down and bottom-up decomposition scenarios. The decomposition algorithm using MAJ combined with other ordinary operators is presented in Section V. We further demonstrate the improved exact-synthesis-aware logic rewriting in Section VI. Then, we perform several experiments over selected Boolean functions as well as EPFL/ISCAS benchmark suites and compare them to the state-of-the-art in Section VII. Finally, we conclude this paper in Section VIII.

II. Related Work

DSD computation is a classical research subject of switching theory [21]. Since the first framework developed in the 1950s [22], the theory has quickly been implemented in the area of digital circuit synthesis by Curtis [23] and Karp [24]. In the past decades, research concentrated on decomposition algorithms and applying them to practical problems, such as circuit restructuring and technology mapping [25]–[28].

The decomposition algorithms are developed based on different logic synthesis representation structures. Ashenhurst introduced an algorithm to detect all the simple decompositions based on decomposition charts, which is effective up to six variables [22]. To reduce the computational

complexity, Bertacco *et al.* [21], [29] used binary decision diagrams (BDDs) to perform DSD. Their method can generate a very compact, canonical multilevel circuit directly from a BDD representation. The BDD-based logic decomposition tool, BDS [30], supports all types of decomposition operations, including AND/OR/XOR and MUX. Minato and De Micheli [31] presented a method to extract all simple disjunctive decompositions by generating an irredundant sum-of-products (ISOPs) and applying factorization, which requires the Minato-Morreale algorithm to produce the ISOP. Recently, Mishchenko proposed the use of DSD structures to efficiently manipulate Boolean functions represented as truth tables and applying it to LUT mapping. Additionally, the MUX is considered as a basic type of decomposition operation [25], [32]. Callegaro *et al.* [33] proposed a bottom-up DSD approach based on cofactors and Boolean difference analysis. The results demonstrated that it is faster compared to the state-of-the-art decomposition strategies. However, these works do not manipulate majority logic, thus missing further optimization opportunities in both XOR/MAJ intensive circuits and nanotechnology circuits which have the MAJ gate as a primitive.

The early attempts to achieve MAJ logic decomposition in 1960s were concerned with the existence of MAJ decomposition based on truth tables or Karnaugh maps [34], [35]. Due to the intractable complexity of the algorithm, it failed to gain interest in automated logic synthesis. Known recent decomposition algorithms that yield MAJ operation are mostly based on BDDs [3], [19], [36]. A constructive library-aware multilevel logic synthesis approach using MAJ as one primitive library is proposed in [19]. The synthesis flow is later applied in the resynthesis loop under tight industry constraints [36]. Another work tries to decompose a function f as $\langle f_a f_b f_c \rangle$, where f_a, f_b, and f_c are pairwise orthogonal erroneous versions of f, by a constructive method using BDDs [3]. However, the solution space is large and one needs high-effort computation to construct MAJ. The key difference of this paper is that: 1) functional decomposition using MAJ with one disjoint support is considered instead of library-aware decomposition and 2) truth-tables and a tree-like data structure (Section III-B) are used to manipulate operations instead of BDDs.

III. Notation and Definitions

A. Boolean Functions

A single-output completely specified Boolean function of n variables is a mapping $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} \in \{0, 1\}$. More generally, a multioutput Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ maps n Boolean input values to m Boolean output values. Boolean variables in an expression can either appear in positive form x_i or in complemented form $\bar{x}_i$.

Given a set of Boolean variables $X = \{x_1, \ldots, x_n\}$, a Boolean function $f(X)$ can be represented by its truth table which is a 2^n size bitstring $(b_{2^n-1} \ldots b_0)_2$, where $i \in [0, 2^n-1]$ is the bit position in the truth table and corresponds to the respective input assignment to X. We write $f = (b_{2^n-1} \ldots b_0)_2$ to mean that the function of f is obtained from the truth table.

Example 1: The truth table of the 3-input parity function $x_1 \oplus x_2 \oplus x_3$ is $f = (1001\ 0110)_2$ or 0x96 in hexadecimal encoding.

Functions can be represented by truth tables or symbolic representations, such as BDDs or SAT formulas. For functions with up to 16 inputs, BDDs have almost never an advantage to the truth table representation since the overhead for constructing the BDD and setting up all data structures outweighs the benefit of the compact representation. In contrast, symbolic representations are beneficial for larger function sizes. Therefore, truth tables are adopted to represent functions with up to 16 inputs.

The support S_f of f is the set of Boolean variables $x_i \in X$ that have an impact on the output value of f (see [37]). The support size $|S_f|$ is the number of its elements. Two functions g and h are called disjoint-support if they share no support variables, i.e., $S_g \cap S_h = \emptyset$.

The positive cofactor of $f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$ with respect to variable x_i is $f_{x_i} = f(x_1, \ldots, 1, \ldots, x_j, \ldots, x_n)$, and the negative cofactor is $f_{\bar{x}_i} = f(x_1, \ldots, 0, \ldots, x_j, \ldots, x_n)$. The identity

$$(fg)_{x_i} = f_{x_i} g_{x_i} \tag{1}$$

is used to calculate cofactors of a product function. A cube cofactor operation is defined as the application of cofactors with respect to different variables recursively, e.g., $f_{x_i \bar{x}_j} = f(x_1, \ldots, 1, \ldots, 0, \ldots, x_n)$. Given two functions $f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$ and $z(x_i, x_j)$, are the generalized positive and negative cofactors of f with respect to function z are $f_z = f(x_1, \ldots, z = 1, \ldots, x_n)$ and $f_{\bar{z}} = f(x_1, \ldots, z = 0, \ldots, x_n)$, respectively. Note that the generalized cofactors are not unique, take f_z as an example, it satisfies $fz \subseteq f_z \subseteq f + \bar{z}$.

The Boolean difference of f with respect to variable x_i is $\partial f / \partial x_i = f_{x_i} \oplus f_{\bar{x}_i}$ [38]. In general, the Boolean difference of f with respect to variable set $X = \{x_1, x_2, \ldots, x_n\}$ is denoted by

$$\frac{\partial}{\partial x_1}\left(\frac{\partial}{\partial x_2} \cdots \left(\frac{\partial f}{\partial x_n}\right)\right) = \frac{\partial f}{\partial x_1 x_2 \ldots x_n}. \tag{2}$$

Given a function $f(X, Y) = h(g(X), Y))$, where $X \cap Y = \emptyset$, the Boolean difference with respect to a variable $x_i \in X$ can be obtained through the Boolean chain rule formulation, which is

$$\frac{\partial f}{\partial x_i} = \frac{\partial h}{\partial g} \frac{\partial g}{\partial x_i} \tag{3}$$

where $\partial h / \partial g$ is the Boolean difference of h with respect to the subfunction g.

The basic Boolean operations considered in this paper are AND, OR, XOR, NOT, and MAJ. The MAJ can be expressed in disjunctive, conjunctive normal form, and exclusive-or sum-of-products (ESOPs) form as

$$\begin{aligned} \langle xyz \rangle &= xy + xz + yz = (x + y)(x + z)(y + z) \\ &= xy \oplus xz \oplus yz \end{aligned} \tag{4}$$

where "$\oplus$" is the XOR operation

$$x \oplus y = x\bar{y} + \bar{x}y = (x + y)(\bar{x} + \bar{y}). \tag{5}$$

The MAJ operation is more expressive than AND/OR and includes them as special cases: $\langle 0xy \rangle = xy$ and $\langle 1xy \rangle = x + y$.

B. Logic Representations

Typically, multilevel logic networks are represented as directed acyclic graphs, called DAGs, in which terminal nodes are input variables or constants, internal nodes are logic operations, and each internal node could potentially be the output for multioutput networks. Homogeneous logic networks, which restrict the nodes' functions to be from a small set of functions, have attracted more interest due to their simplicity and thus optimization opportunities. Popular instances of homogeneous logic representations include NAND and NOR circuits [39], AIGs [40], and recently proposed MIGs [9]. XMGs are an extension of MIGs which additionally use the XOR primitive.

A DSD structure is a data structure to manage the computational operations of decomposition/composition functions, which was first introduced by Mishchenko and Brayton [32]. In this paper, to obtain an XMG, we use a modified DSD structure, referred to as mDSD structure, to manage the computational operations. The critical difference is that the mDSD structure is an extension of the DSD structure by introducing the MAJ operator. An mDSD structure over the primary input variables $X = \{x_1, \ldots, x_n\}$ is a DAG $T = (V, E, Y)$ with:

1) a finite set of nodes $V = X \cup G$, where G are internal nodes representing the logic operations in the tree;
2) a finite multiset of edges $E \in G \times (V \times \mathbb{B})$, where the first element in the tuple is a source node and the second element is a pair of a target node and a polarity bit to indicate the edge complemented attribute;
3) and a finite multiset of outputs $Y \in V \times \mathbb{B}$.

For each internal node in G, the operations can be basic gates (AND, OR, XOR, and MAJ) or prime nodes. Each prime node is associated with a truth table, which indicates it needs further functional decomposition. A function is called prime function if it cannot be disjointly decomposed by any of the basic gates. The number of inputs to internal nodes depends on their operation types. AND, OR, and XOR nodes have two inputs, MAJ nodes have three inputs, and prime nodes can have multiple ordered inputs. If the resulting mDSD structure has no prime nodes, it is isomorphic to an XMG, as it can be directly derived from it. Consequently, the proposed logic decomposition algorithm enables further opportunities in the synthesis with XMG logic networks.

Example 2: Fig. 3 shows an example of the DSD structure and its corresponding mDSD structure after one step majority decomposition. The function with truth table 0xE2EE represented by DSD structure shown in Fig. 3(a) contains one prime node which cannot be disjointly decomposed by any operations using AND, OR, and XOR. After introducing MAJ operator, we can write $f = \langle \bar{x}_1 g h \rangle$, where $g = x_2 \oplus x_3$ and $h = \texttt{PRIME}(x_2, x_3, x_4)$. As shown in Fig. 3(b), the mDSD structure contains a prime node with truth table 0xCA, where dashed lines indicate the complemented attribute.

C. Disjoint-Support Decomposition

The decomposition of a logic function $f(x_1, \ldots, x_n)$ identifies a set of functions $a_i(x_i)$ with no shared input variables, and a function L [21] such that

$$f = L(a_0, \ldots, a_i, \ldots, a_n).$$

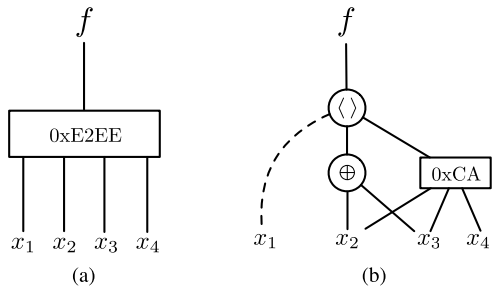

Fig. 3. (a) DSD structure represented function and (b) its corresponding mDSD structure after one step majority decomposition.

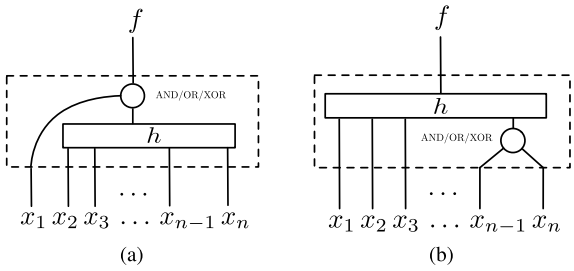

Fig. 4. (a) Top-down DSD. (b) Bottom-up DSD.

The DSD is a special case of Boolean logic decomposition. Function f has a disjoint decomposition when the two other functions, say g and h, satisfy the equation

$$f(X) = h(X_1, g(X_2)),\ X_1 \cup X_2 = X\ \ X_1 \cap X_2 = \emptyset. \quad (6)$$

Given a set of operations G, a logic function is called *full-DSD* if it can be represented by operations in G with disjoint supports. A function is *non-DSD* if it is a prime function without any disjoint support. A function is *partial-DSD* if it can be represented as the combination of operations in G with disjoint supports and prime functions.

Example 3: Given a set of operations G, which consists of 2-input OR (AND), and XOR, the logic function $f = x_1 \oplus x_2 x_3$ is full-DSD, while $g = \texttt{PRIME}(x_2, x_3, x_4)$ with truth table 0xCA is non-DSD, since it cannot be represented by any disjoint support expression using operations in G. In contrast, the function $h = x_1 \oplus g$ is partial-DSD.

DSD can be implemented from both top-down (from outputs to inputs) and bottom-up (from inputs to outputs) decomposition scenarios. As show in Fig. 4(a), the top-down DSD writes the function $f(x_1, \ldots, x_n)$ as $f = x_1 \circ h(x_2, \ldots, x_n)$, where $\circ$ is one of the AND/OR/XOR operations. In contrast, Fig. 4(b) indicates the bottom-up DSD that writes the function $f(x_1, \ldots, x_n)$ as $f = h(x_1, \ldots, x_{n-2}, x_{n-1} \circ x_n)$.

D. Exact Synthesis

Exact synthesis is the task of finding an optimum logic network representation for a given input specification with respect to some cost criteria [41]. For instance, size optimization is to find a network with the smallest number of nodes, while depth optimization is to find a network with the smallest number of logic levels. Recent advances in the implementation of exact synthesis algorithms make the SAT-based exact synthesis an essential engine in logic

synthesis framework. SAT formulas are generated based on different underlying logic representations. They are distinct in the number of encoding variables and clauses, each having their own tradeoffs. We refer the reader to [42]–[44] for detailed description.

In terms of runtime, most of the overall runtime is typically required to prove an instance is unsatisfiable for small r. If r is large enough, the solver will return the solution, if the instance is satisfiable. Generally, a timeout value t is given to ask whether the SAT solver can find an optimum solution within t seconds. If not, the algorithm terminates, and other strategies are required. Although exact synthesis is efficient for small functions (having up to six variables), it can also be implemented for large functions when being applied to small subnetworks to guarantee local optimality [17].

E. NPN Classification

Two functions are NPN-equivalent, if one of them can be obtained from the other by negating inputs, permuting inputs, or negating the output [45].

Example 4: Given two functions $f = ab+c$ and $g = \bar{a}+b\bar{c}$, they are NPN-equivalent because f can be transformed into g by swapping variables a and c, while both of a and c are negated.

All 2^{2^n} Boolean functions over n variables can be partitioned into 2, 4, 14, 222, 616,126 NPN classes for $n = 1,\ 2,\ 3,\ 4,\ 5$, while 200, 253, 952, 527, 185 NPN classes are needed for $n = 6$ [46].

IV. Majority Logic Decomposition

This section presents our method to implement MAJ logic decomposition, from both top-down and bottom-up decomposition scenarios.

A. Top-Down Decomposition

1) Conditions: This paper makes use of functional decomposition based on the MAJ operation. The aim is to represent $f(x_1, \ldots, x_n)$ as $\langle zgh\rangle$ such that g and h independent from subfunction z. We make use of the following decomposition [35].

Theorem 1: Let f be a Boolean function and z a subfunction with support $S_z \subseteq S_f$. Then

$$f = \langle zf_z f_{\bar{z}}\rangle \qquad \text{if, and only if } \bar{f}_z f_{\bar{z}} = 0 \tag{7}$$

where f_z and $f_{\bar{z}}$ are the generalized positive and negative cofactors of function f with respect to function z, respectively.

Proof: From Shannon's decomposition (in XOR form), we know that $f = zf_z \oplus \bar{z}f_{\bar{z}}$. Also we have $\langle zf_z f_{\bar{z}}\rangle = zf_z \oplus zf_{\bar{z}} \oplus f_z f_{\bar{z}}$ [see (4)]. We check for which condition these two equations differ

$$\begin{aligned} &zf_z \oplus \bar{z}f_{\bar{z}} \oplus zf_z \oplus zf_{\bar{z}} \oplus f_z f_{\bar{z}} \\ &\overset{a\oplus a=0}{=} \bar{z}f_{\bar{z}} \oplus zf_{\bar{z}} \oplus f_z f_{\bar{z}} \\ &\overset{\bar{a}b\oplus ab=b}{=} f_{\bar{z}} \oplus f_z f_{\bar{z}} \overset{a\oplus ab=a\bar{b}}{=} \bar{f}_z f_{\bar{z}}. \end{aligned}$$

Hence, the equations only differ when $\bar{f}_z f_{\bar{z}} = 1$. Because $\bar{f}_z f_{\bar{z}} = 0$, $\bar{f}_z$ and $f_{\bar{z}}$ have an empty intersection. Therefore, we can write $f = \langle zf_z f_{\bar{z}}\rangle$. ∎

The subfunction z could be either with a single variable or multiple variables. The cofactors with respect to a single variable are unique, but are not uniquely determined for multiple variables in general [47].

Example 5: Given two functions that are $f(x_1, x_2, x_3, x_4)$ and $z(x_1, x_2) = x_1 + x_2$, the generalized positive cofactor of f with respect to z is $f_z = f(z = 1, x_3, x_4)$. Because the input assignments $\{x_1 = 0, x_2 = 1\}$, $\{x_1 = 1, x_2 = 0\}$, and $\{x_1 = 1, x_2 = 1\}$ for z produce the same output, therefore f_z can be obtained by any one of the following three cube cofactors:

$$\begin{aligned} f_{\bar{x}_1 x_2} &= f(0, 1, x_3, x_4) \\ f_{x_1 \bar{x}_2} &= f(1, 0, x_3, x_4) \\ f_{x_1 x_2} &= f(1, 1, x_3, x_4). \end{aligned}$$

However, these three cube cofactors may not be equal and thus f_z is not unique. In contrast, $f_{\bar{z}} = f(z = 0, x_3, x_4)$ is unique as only the input assignment $\{x_1 = 0, x_2 = 0\}$ makes $z = 0$, thus $f_{\bar{z}} = f(0, 0, x_3, x_4)$.

To get the unique cofactors of the subfunctions with multiple variables, the functional conditions of each candidate should be checked. The functional conditions are derived from the logical operations. The subfunction candidates with multiple variables considered are 2-input OR (AND), 2-input XOR, and 3-input MAJ to factorize an XMG. Concerning the OR operation of two variables x_1 and x_2 as a subfunction, the functional conditions are

$$f_{\bar{x}_1 x_2} = f_{x_1 \bar{x}_2} = f_{x_1 x_2}. \tag{8}$$

Similarly, the function conditions using $\langle x_1 x_2 x_3\rangle$ and $x_1 \oplus x_2$ as subfunctions should be

$$\begin{aligned} f_{x_1 x_2 x_3} &= f_{x_1 x_2 \bar{x}_3} = f_{x_1 \bar{x}_2 x_3} = f_{\bar{x}_1 x_2 x_3} \\ f_{\bar{x}_1 \bar{x}_2 \bar{x}_3} &= f_{\bar{x}_1 \bar{x}_2 x_3} = f_{\bar{x}_1 x_2 \bar{x}_3} = f_{x_1 \bar{x}_2 \bar{x}_3} \end{aligned} \tag{9}$$

and

$$\begin{aligned} f_{x_1 x_2} &= f_{\bar{x}_1 \bar{x}_2} \\ f_{\bar{x}_1 x_2} &= f_{x_1 \bar{x}_2} \end{aligned} \tag{10}$$

respectively.

Example 6: Given a truth table $f = (1111\,1110\,1110\,0000)_2$ or `0xFEE0` representing the function $f(x_1, x_2, x_3, x_4)$, to check whether it can be written as $f = \langle zgh\rangle$, where $z = x_1 + x_2$, we first check the functional conditions of the subfunction z. The cube cofactors are calculated as follows:

$$\begin{aligned} f_{\bar{x}_1 x_2} &= f(0, 1, x_3, x_4) = (1111\,1111\,1111\,0000)_2 \\ f_{x_1 \bar{x}_2} &= f(1, 0, x_3, x_4) = (1111\,1111\,1111\,0000)_2 \\ f_{x_1 x_2} &= f(1, 1, x_3, x_4) = (1111\,1111\,1111\,0000)_2. \end{aligned}$$

Hence, (8) holds. Then we calculate f_z and $f_{\bar{z}}$ as Example 5 does

$$\begin{aligned} f_z &= f(z = 1, x_3, x_4) = (1111\,1111\,1111\,0000)_2 \\ f_{\bar{z}} &= f(z = 0, x_3, x_4) = (1111\,0000\,0000\,0000)_2. \end{aligned}$$

One can verify that $\bar{f}_z f_{\bar{z}} = 0$, thus (7) holds and we can write $f = \langle zf_z f_{\bar{z}}\rangle$.

2) Cofactor Optimization: Due to satisfiability do not care (SDC) conditions generated by the function between inputs with shared support [47], it is possible to minimize the subfunctions while maintaining the function validity. For our case, the subfunction optimization is applied by exploiting the MAJ operator functionality, which is on the basis of

$$f = \langle zgh \rangle = \begin{cases} z = g & \text{if } z = g \\ h & \text{if } z \neq g. \end{cases} \tag{11}$$

The optimization opportunity arises from the second case, which means that the output value is determined by h if z and g assume the opposite logic value. In this case, the actual values of z and g are not of interest, which opens up optimization opportunity to balance the support size of z and g to reduce the decomposition complexity. Therefore, the optimization can be conducted by cyclic optimization over all subfunction combinations that are (z, g), (z, h), and (g, h). Since our MAJ decomposition $f = \langle zgh \rangle$ makes g and h not depend on the subfunction z, we just need to balance the support size of g and h.

Next, we present a cofactor optimization method by encoding the problem as an instance of the satisfiability modulo theories (SMTs) problem. Modern SMT solvers support bitvectors of arbitrary size, which is inherently suitable for truth table representations. For clarity, we consider an n-variable Boolean function $f = \langle zf_z f_{\bar{z}} \rangle$, where z is a subfunction with either multiple variables or a single variable. We introduce two bitvectors $g = (g_{2^n-1} \dots g_0)_2$ and $h = (h_{2^n-1} \dots h_0)_2$ with size 2^n to represent the two cofactors f_z and $f_{\bar{z}}$, respectively, where $i \in [0, 2^n - 1]$ is the bit position of the bitvectors. Note that we already have truth tables of z and f, our aim is to minimize the total support size of g and h while maintaining the MAJ functionality. Recalling the basis presented in (11), we can derive the following properties.

1) If the logic values of the ith bit of z and f are opposite, then g_i and h_i are determined by the logic value of f

$$(z_i \neq f_i) \rightarrow (g_i = h_i = f_i). \tag{12}$$

2) If the logic values of the ith bit of z and f are the same, then

$$(z_i = f_i = 1) \rightarrow (g_i | h_i = 1) \tag{13}$$

$$(z_i = f_i = 0) \rightarrow (g_i \& h_i = 0) \tag{14}$$

where "|" and "&" are binary OR and AND operations, respectively, bitwise.

Example 7: Let $f = \langle x_1 x_2 x_3 \rangle$ (truth table `0xE8`) and $z = x_1$. The decomposition algorithm without cofactors optimization produces the result $f = \langle x_1 f_{x_1} f_{\bar{x}_1} \rangle$, where $f_{x_1} = \langle 1 x_2 x_3 \rangle$ and $f_{\bar{x}_1} = \langle 0 x_2 x_3 \rangle$. The total support size of cofactors is $|S_{f_{x_1}}| + |S_{f_{\bar{x}_1}}| = 4$. We demonstrate how to optimize the cofactors $g = f_{x_1}$ and $h = f_{\bar{x}_1}$ using SMT. The truth tables are the following.

g	h	z	f	g	h	z	f
g_7	h_7	1	1	g_6	h_6	0	1
g_5	h_5	1	1	g_4	h_4	0	0
g_3	h_3	1	1	g_2	h_2	0	0
g_1	h_1	1	0	g_0	h_0	0	0

According to (12)–(14), we write the constraints

$$\begin{aligned} g_7 | h_7 = 1 \qquad & g_6 = h_6 = 1 \\ g_5 | h_5 = 1 \qquad & g_4 \ \& \ h_4 = 0 \\ g_3 | h_3 = 1 \qquad & g_2 \ \& \ h_2 = 0 \\ g_1 = h_1 = 0 \qquad & g_0 \ \& \ h_0 = 0. \end{aligned}$$

All the above constraints are passed to an SMT solver for satisfiability checking. By enumerating all solutions (we add blocking constraints to prevent the solver to generate the same result twice), we calculate the minimal total support size of g and h. Finally, the optimal results are $g = (1111\ 0000)_2$ and $h = (1100\ 1100)_2$. These are the primary inputs x_2 and x_3. Hence, the solution achieves the optimal total support size that is $|S_g| + |S_h| = 2$ and we can write $f = \langle x_1 x_2 x_3 \rangle$.

The computation cost is expensive, since we enumerate all the solutions. For instance, Example 7 has six undefined entries out of eight, and each entry has three possible combinations. Hence, the example has in total $3^6 = 729$ solutions. To make our method scalable, we should speed up the solving time by symmetry breaking to discard equivalent solutions and by exploiting Boolean properties to prevent worse solutions.

In terms of search space pruning, we add constraint

$$g \neq h \tag{15}$$

to indicate the two bitvectors g and h should not be equal. Otherwise, it means $f = \langle zgh \rangle = g = h$, which is the trivial case that we can keep from happening during the majority decomposition.

For symmetry breaking, according to MAJ commutativity law, $\langle zgh \rangle = \langle zhg \rangle$, therefore, if we already get the solution, say $g = X$ and $h = Y$, then we add constraints to discard another solution $g = Y$ and $h = X$

$$(g \neq Y) \vee (h \neq X). \tag{16}$$

To exploit Boolean properties, because our MAJ decomposition produces $\langle zgh \rangle$, g and h do not depend on subfunction z, which opens up an opportunity by adding more constraints on the two bitvectors to prevent worse solutions.

Lemma 1: Given an n-variable function $f(x_1, \dots, x_n) = f(f_{2^n-1}, \dots, f_0)_2$, if f does not depend on variable x_i, $i \in [1, n]$, then there are 2^{n-1} pairs of bits in the truth table having the same logic value.

Proof: As f does not depend on x_i, then $x_i = 0$ or $x_i = 1$ will not change the output value of f. In a 2^n length truth table, there are 2^n input combinations, therefore $2^n \cdot 2^{-1}$ pairs of input combinations produce the same output, which makes 2^{n-1} pairs of bits in the truth table having the same logic value. ■

Corollary 1: If $f(x_1, \dots, x_n)$ does not depend on x_i, where $i \in [1, n]$, assume the 2^{n-1} pairs of bits are $p_j = \{f_{a_j}, f_{b_j}\}$, $j \in [1, 2^{n-1}]$, then the following conditions should be satisfied.

1) Indexes a_j and b_j in all pairs form a set $[0, 2^{n-1}]$, which is consistent with 2^n length truth table. The indices of all pairs are distinct from each other.
2) $a_j + 2^{i-1} = b_j$.

Proof: Since f does not depend on x_i, if two input combinations just differ in x_i, then the output value should be the same. Considering the property of primary input x_i represented by

the truth table, the "0" and "1" logic values are repeated with 2^{i-1} bit positions interval. As an example, the truth tables of the primary inputs x_i where $i \in [1, 3]$ are shown as follows:

$$x_1 = (1010\,1010)_2$$
$$x_2 = (1100\,1100)_2$$
$$x_3 = (1111\,0000)_2.$$

The positions of logic value interval is the direct reason for the output equivalence. If f does not depend on x_1, one can verify that $p_1 = \{f_0, f_1\}$ as the logic values of bit positions 0 and 1 in x_2 and x_3 are all the same. Similarly, the other pairs are $p_2 = \{f_2, f_3\}$, $p_3 = \{f_4, f_5\}$, and $p_4 = \{f_6, f_7\}$, which satisfy the two conditions illustrated above. In terms of x_i, one can get the conclusions in the same way, which concludes the proof. ∎

Example 8: In Example 7, g and h do not depend on $z = x_1$, which is the ith input variable ($i = 0$). Hence, we add following additional constraints:

$$g_0 = g_1 \quad g_2 = g_3 \quad g_4 = g_5 \quad g_6 = g_7$$
$$h_0 = h_1 \quad h_2 = h_3 \quad h_4 = h_5 \quad h_6 = h_7.$$

By symmetry breaking and exploiting Boolean properties, the number of solutions of Example 7 is reduced from 729 to 2, which results in $\langle x_1 x_2 x_3\rangle$ and $\langle x_1 \langle 0 x_2 x_3\rangle \langle 1 x_2 x_3\rangle\rangle$. The former case has smaller total support size, while the latter case is the result before optimization. Finally, we selected $\langle x_1 x_2 x_3\rangle$ as the optimization result.

The proposed cofactor optimization method is outlined in Algorithm 1. Given an mDSD structure $\langle z f_z f_{\bar{z}}\rangle$, we first initialize the SMT solver s and the iteration number *it*. The two bitvector variables g and h with length 2^n are used as the encodings of the cofactors. Next, we add SMT constraints to generate SMT clauses based on (12)–(15) and Corollary 1. The SMT solver enumerates all the solutions by dealing with the clauses until the *it* reaches the limitation number max_iteration. The iteration limitation is defined by considering that the solution space may still be unaffordable for some extreme cases or functions with a large number of variables. Finally, the solution with the minimal total support size is returned as the optimization solution.

3) Complexity Analysis: Given an n-variable Boolean function $f = (x_1, \ldots, x_n)$, in order to estimate the computational complexity of the proposed majority decomposition $f = \langle zgh\rangle$, we first try to construct subfunction z with multiple variables using 2-input AND, OR, and XOR operators, and 3-input MAJ operator. For instance, we can construct the subfunctions as $z = x_1 x_2$, $z = x_1 + x_2$, $z = x_1 \oplus x_2$, and $z = \langle x_1 x_2 x_3\rangle$. Considering all the combinations of input variables, there are $\binom{n}{2}$ cases for 2-input operators, while $\binom{n}{3}$ cases for the MAJ operator. If the majority decomposition using subfunctions with multiple variables is not successful, we then try the single variable functions that are $z = x_1, \ldots, z = x_n$. Therefore, the overall majority decomposition algorithm has a computational complexity of $O(n^3)$. For the subsequent cofactors optimization, the complexity is depended on both the execution of the SMT solver and the frequency of the execution. The worst case is that the solver is executed up to the maximum number of iteration (*max_iteration*) times.

Algorithm 1: Cofactors Optimization Based on SMT

Input : An mDSD structure $\langle z f_z f_{\bar{z}}\rangle$ with n variables
Output: An optimized mDSD structure

```
s ← SMT solver, it ← 0 ;
Declare two bitvectors g and h with length 2^n in s;
Add SMT constraints to s based on(12), (13), (14), (15)
  and Corollary 1;
while s.check() == SAT && it < max_iteration do
    it ← it + 1;
    Evaluate the current solution, say g = X, h = Y, and
      record the best solution;
    Add new constraints to s based on (16) to discard the
      solution with the same quality;
    Add new constraints (g ≠ X) ∨ (h ≠ Y) to s for
      enumerating all the solutions;
end
Return the best solution;
```

B. Bottom-Up DSD

The aim of bottom-up DSD using MAJ is to represent $f(X, Y) = f(x_1, \ldots, x_n)$ as $f = h(g(X), Y)$, where $X = \{x_i, x_j, x_k\}$, $g(X) = \langle x_i x_j x_k\rangle$, and $X \cap Y = \emptyset$. The necessary and sufficient conditions to perform MAJ bottom-up DSD were presented in [48]. We present a strategy based on cofactors and Boolean difference to obtain such composition functions, which is useful to update truth tables during the decomposition. For clarity, we first revisit the theorem proposed in [48].

Theorem 2: Let $f(X, Y)$ be a Boolean function with $X = \{x_i^*, x_j^*, x_k^*\}$ where $x_\kappa^* = x_\kappa$ or $\bar{x}_\kappa$ for $\kappa \in \{i, j, k\}$, $X \cap Y = \emptyset$. There exists a function $h(g(X), Y) = f(X, Y)$, where $g(X) = \langle x_i^* x_j^* x_k^*\rangle$, if and only if the following conditions are satisfied.

Condition 1:

$$\frac{\partial f}{\partial x_i x_j} = \frac{\partial f}{\partial x_i x_k} = \frac{\partial f}{\partial x_j x_k}. \tag{17}$$

Condition 2:

$$\left(\frac{\partial f}{\partial x_i}\right)_{\bar{x}_j \bar{x}_k} = a_i,\ \left(\frac{\partial f}{\partial x_j}\right)_{\bar{x}_i \bar{x}_k} = a_j,\ \left(\frac{\partial f}{\partial x_k}\right)_{\bar{x}_i \bar{x}_j} = a_k \tag{18}$$

such that either all three of a_i, a_j, and a_k are constant 0, or two of a_i, a_j, and a_k are equal and the third is constant 0.

Proof: See [48]. ∎

Corollary 2: If there exists a function $h(g, Y)$, where $g(X) = \langle x_i^* x_j^* x_k^*\rangle$, such that $f(X, Y) = h(g(X), Y)$, $X \cap Y = \emptyset$, then the decomposition function $h(g, Y)$ can be obtained as follows:

$$h(g, Y) = f_{\bar{x}_i^* \bar{x}_j^*} \oplus g\left(\frac{\partial f}{\partial x_i}\right)_{\bar{x}_j^* x_k^*}. \tag{19}$$

Proof: The Boolean function f can be decomposed using positive Davio expansion with respect to x_i, which is

$$f = f_{\bar{x}_i} \oplus x_i \frac{\partial f}{\partial x_i}. \tag{20}$$

By applying the positive Davio expansion of h with respect to g, we obtain

$$h(g, Y) = h_{\bar{g}} \oplus g\frac{\partial h}{\partial g}. \tag{21}$$

In terms of $h_{\bar{g}}$, it can be obtained by finding an assignment for x_i^*, x_j^*, and x_k^*, such that $g(X) = \langle x_i^* x_j^* x_k^* \rangle = 0$, since the output of MAJ is dominated by two of its inputs, then all the three assignments, $\{x_i^* = 0, x_j^* = 0\}$, $\{x_i^* = 0, x_k^* = 0\}$, and $\{x_j^* = 0, x_k^* = 0\}$ are satisfiable. Then it follows that:

$$h_{\bar{g}} = h(0, Y) = f_{\bar{x}_i^* \bar{x}_j^*}. \tag{22}$$

To calculate $\partial h/\partial g$, we first obtain the Boolean difference of g with respect to x_i by definition and simplification

$$\begin{aligned}\frac{\partial g}{\partial x_i} &= g_{x_i} \oplus g_{\bar{x}_i} \\ &= \left(x_j^* + x_k^* + x_j^* x_k^*\right) \oplus x_j^* x_k^* \\ &= x_j^* \oplus x_k^*. \end{aligned} \tag{23}$$

Hence, if $x_j^* \oplus x_k^* = 1$ holds, e.g., $x_j^* = 0$ and $x_k^* = 1$, then $\partial g/\partial x_i = 1$. Since $f(X, Y) = h(g(x_i^*, x_j^*, x_k^*), Y)$, according to the Boolean chain rule shown in (3) and the identity presented in (1), we obtain

$$\begin{aligned}\left(\frac{\partial f}{\partial x_i}\right)_{\bar{x}_j^* x_k^*} &= \left(\frac{\partial h}{\partial g}\right)_{\bar{x}_j^* x_k^*} \left(\frac{\partial g}{\partial x_i}\right)_{\bar{x}_j^* x_k^*} \\ &= \left(\frac{\partial h}{\partial g}\right)_{\bar{x}_j^* x_k^*} 1. \end{aligned} \tag{24}$$

As $\partial h/\partial g$ is not supported by x_j nor by x_k, therefore

$$\frac{\partial h}{\partial g} = \left(\frac{\partial h}{\partial x_i}\right)_{\bar{x}_j^* x_k^*} = \left(\frac{\partial f}{\partial x_i}\right)_{\bar{x}_j^* x_k^*}. \tag{25}$$

By substitution of (22) and (25) into (21), the proof is concluded. Due to the symmetry properties of MAJ operator, note that (22) can be also written as

$$h_{\bar{g}} = h(0, Y) = f_{\bar{x}_i^* \bar{x}_k^*} = f_{\bar{x}_j^* \bar{x}_k^*} \tag{26}$$

and (25) can be also written as

$$\frac{\partial h}{\partial g} = \left(\frac{\partial h}{\partial x_j}\right)_{\bar{x}_i^* x_k^*} = \left(\frac{\partial f}{\partial x_j}\right)_{x_i^* \bar{x}_k^*} \tag{27}$$

$$= \left(\frac{\partial h}{\partial x_k}\right)_{\bar{x}_i^* x_j^*} = \left(\frac{\partial f}{\partial x_k}\right)_{x_i^* \bar{x}_j^*}. \tag{28}$$

■

Corollary 3: If Theorem 2 is satisfied for variables x_i^*, x_j^*, x_k^*, and in Condition 2 $a_\kappa = 0$, then x_δ^* and x_σ^* have the same polarities, where $\kappa \in I = \{i, j, k\}$, $\{\delta, \sigma\} \in I$, $\delta \neq \sigma \neq \kappa$. Otherwise, if $a_\kappa \neq 0$, x_δ^* and x_σ^* have the opposite polarities.

Proof: Without loss of generality, we take a_i as an example, assume $f(X, Y) = h(g(X), Y)$ and $g(X) = \langle x_i^* x_j^* x_k^* \rangle$, then according to (3)

$$a_i = \left(\frac{\partial f}{\partial x_i}\right)_{\bar{x}_j^* \bar{x}_k^*} = \left(\frac{\partial f}{\partial x_i}\right)_{x_j^* = 0, x_k^* = 0} \tag{29}$$

$$= \left(\frac{\partial f}{\partial g}\frac{\partial g}{\partial x_i}\right)_{x_j^* = 0, x_k^* = 0} \tag{30}$$

$$= \left(\frac{\partial f}{\partial g}\left(x_j^* \oplus x_k^*\right)\right)_{x_j^* = 0, x_k^* = 0}. \tag{31}$$

Since $\partial f/\partial g$ is not zero, if $a_i = 0$, it indicates that $x_j^* \oplus x_k^* = 0$, then both $\{x_j, x_k\}$ and $\{\bar{x}_j, \bar{x}_k\}$ are acceptable. Therefore, if $a_i = 0$, x_j^* and x_k^* have the same polarities. Otherwise, if $a_i \neq 0$, it indicates that $x_j^* \oplus x_k^* = 1$, then both $\{\bar{x}_j, x_k\}$ and $\{x_j, \bar{x}_k\}$ are acceptable. Hence, x_j^* and x_k^* have the opposite polarities. The conclusion can be easily applied to a_j and a_k. ■

Example 9: Given a function $f(x_1, x_2, x_3, x_4) = \bar{x}_1 x_2 + x_2 x_3 + \bar{x}_1 x_3 + x_4$, to check whether it can be decomposed as $f = h(\langle x_1^* x_2^* x_3^* \rangle, x_4)$, we first check the condition presented in (17)

$$\begin{aligned}\frac{\partial f}{\partial x_1 x_2} &= \frac{\partial}{\partial x_1}\left(\frac{\partial f}{\partial x_2}\right) = \frac{\partial}{\partial x_1}\left(f_{x_2} \oplus f_{\bar{x}_2}\right) \\ &= \frac{\partial}{\partial x_1}((\bar{x}_1 \oplus x_3)\bar{x}_4) = \bar{x}_4. \end{aligned} \tag{32}$$

Also, we can compute $\partial f/\partial x_1 x_3$ and $\partial f/\partial x_2 x_3$ in the same way, and (17) holds

$$\frac{\partial f}{\partial x_1 x_2} = \frac{\partial f}{\partial x_1 x_3} = \frac{\partial f}{\partial x_2 x_3} = \bar{x}_4. \tag{33}$$

To check Condition 2, we compute

$$\begin{aligned}&\frac{\partial f}{\partial x_1}(x_2 = 0, x_3 = 0) = a_1 = 0 \\ &\frac{\partial f}{\partial x_2}(x_1 = 0, x_3 = 0) = a_2 = \bar{x}_4 \\ &\frac{\partial f}{\partial x_3}(x_1 = 0, x_2 = 0) = a_3 = \left(\overline{x_1 \oplus x_2}\right)\bar{x}_4|_{x_1 = x_2 = 00} = \bar{x}_4. \end{aligned}$$

Therefore, Condition 2 is satisfied as $a_2 = a_3$ and $a_1 = 0$. We can write $f = h(\langle x_1^* x_2^* x_3^* \rangle, x_4)$. The polarities of the x_1^*, x_2^*, and x_3^* are determined by Corollary 3. Because $a_1 = 0$, then x_2^* and x_3^* have the same polarities, which are both distinct from x_1^*, then both $\{x_1, \bar{x}_2, \bar{x}_3\}$ or $\{\bar{x}_1, x_2, x_3\}$ are acceptable polarities. Take the former case as an example, the composition function h can be obtained based on (19)

$$\begin{aligned}h(g, Y) &= f_{\bar{x}_1^* \bar{x}_2^*} \oplus g\left(\frac{\partial f}{\partial x_1}\right)_{\bar{x}_2^* x_3^*} \\ &= f_{\bar{x}_1 x_2} \oplus g\left(\frac{\partial f}{\partial x_1}\right)_{x_2 \bar{x}_3} \\ &= 1 \oplus g\bar{x}_4 = \overline{g\bar{x}_4} = \bar{g} + x_4 = \overline{\langle x_1 \bar{x}_2 \bar{x}_3 \rangle} + x_4. \end{aligned} \tag{34}$$

For the latter case, one can verify that $h(g, Y) = \langle \bar{x}_1 x_2 x_3 \rangle + x_4$, complies with the inversion propagation rule of MAJ, that is $\langle \bar{x}_1 x_2 x_3 \rangle = \overline{\langle x_1 \bar{x}_2 \bar{x}_3 \rangle}$.

V. Functional Decomposition Using Majority

This section describes our proposed functional decomposition algorithm. The input to the algorithm is a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ represented as a truth table. The output is an mDSD structure, which is isomorphic to an XMG.

The algorithm is outlined in Algorithm 2. Given a Boolean function f, an mDSD structure is initialized with just one prime node, which is associated with the truth table of f. Then, the algorithm uses *recursive_decomp*(f) for an exhaustive evaluation of all prime nodes until the nodes cannot be decomposed. The computational results are manipulated by the mDSD structure.

Algorithm 2: Functional Decomposition

Input : A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$
Output: An mDSD structure R

```
1  X: the set of all input variables of f;
2  X_m: the set of input variables in X that satisfy (7);
3  function recursive_decomp (f)
4      R ←mDSD_initialize(f);
5      repeat
6          R ←func_decomp (R);
7      until exist_no_prime_node(R);
8      return R;
9  function func_decomp (R)
10     for each prime node n in R do
11         h ←truth table of n;
12         y_s ← min(|S_{f_{x_i}}| + |S_{f_{x̄_i}}|), x_i ∈ X;
13         y_m ← min(|S_{f_{x_j}}| + |S_{f_{x̄_j}}|), x_j ∈ X_m;
14         if is_basic(h) then update_basic(n,R);
15         else if is_DSD(h) then update_DSD(n,R);
16         else if |S_f| ≤ l then update_exact(n,R);
17         else if h satisfy (7), h = ⟨z f_z f_z̄⟩ then
18             if (|S_z| > 1) or (|S_z| == 1 && y_m ≤ y_s) then
                   update_MAJ(n,R);
19         else update_Shannon(n,R);
20     end
21     return R;
```

The decomposability of the prime node is checked according to the following conditions sequentially by func_decomp(R). Given a set of input variables X, we record the variables that satisfy Boolean properties (7) as X_m. Initially, we obtain the truth table of the prime node (line 11) and calculate the minimum total support size of cofactors of X (line 12) and X_m (line 13), which are recorded as y_s and y_m, respectively.

1) If the truth table equals the basic functions, such as two-input AND, OR, and XOR operators, and three-input MAJ operator, then the prime node will be replaced by these basic gates (line 14).
2) For each input of the prime node, we traverse all input variables to check whether it can be DSD-decomposed (line 15). Note that the bottom-up MAJ DSD described in Section IV-B is implemented.
3) If no DSD exists, the prime node must be a prime function, if the prime function has up to l variables, we call the exact synthesis routine described in [41] (line 16).
4) If the three previous conditions do not hold, we try the MAJ top-down decomposition demonstrated in Theorem 1 to exploit the opportunity of rewriting $f = \langle z f_z f_{\bar{z}} \rangle$ (line 17). Note that we first try to construct the subfunction with multiple variables. If unsuccessful, we then try the single-variable subfunctions. As the single-variable subfunction that satisfies (7) may not be unique, we also consider a heuristic approach to make a selection between majority logic decomposition and the subsequent Shannon decomposition. This is motivated by the support-reducing technique which is effective to control the subsequent size and depth of the mDSD structure. Hence, we add the additional condition $y_m \leq y_s$ for majority logic decomposition using single-variable subfunctions (line 18).
5) If all the above conditions do not hold, Shannon decomposition is applied (line 19).

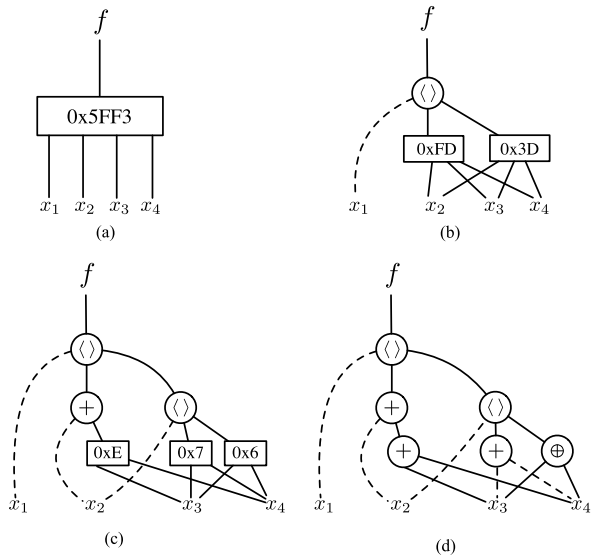

Fig. 5. Decomposition example manipulated by mDSD structure. (a) Initial mDSD structure. (b) MAJ top-down decomposition. (c) DSD using OR and MAJ top-down decomposition. (d) Replace prime nodes by basic gates.

Note that when any of these conditions is satisfied, we create a corresponding operator node and new prime nodes with updated truth tables and support sets. Finally, the resulting mDSD structure, which is isomorphic to an XMG due to the introduction of Shannon decomposition, is returned as the solution.

The conditions and updating schemes for MAJ decompositions are demonstrated in Section IV. For the conditions and updating schemes of DSD, readers can refer to [21] and [33] for further details.

Example 10: Given the function `0x5FF3`, the steps of the decomposition are shown in Fig. 5. We initialize the mDSD structure in Fig. 5(a), then we perform the MAJ top-down decomposition and create two new prime nodes with updated truth tables and support sets in Fig. 5(b). The decomposition continues to decompose the two prime nodes using DSD with the OR operator and MAJ top-down decomposition in Fig. 5(c). Finally, as the truth tables of the three prime nodes are consistent with basic gates, we obtain the final mDSD structure in Fig. 5(d), which is isomorphic to an XMG.

VI. LUT-Based Exact Synthesis

In this section, we describe the application scenario to LUT-based size optimization, in which logic decomposition can be employed.

A. Brief Review of LUT-Based Optimization

LUT-based mapping is a special case of technology mapping in which logic networks are covered by k-input LUTs (k-LUTs). It provides an attractive way to identify the subnetworks. Given an input network N, the approach proposed

in [12] first maps the network into k-LUTs, e.g., in 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 in a database that stores the optimum representations of the NPN classes, which is referred to as Boolean function mining. Finally, the locally optimum networks are merged to construct an optimized, functionally equivalent network N'. The optimization process may be iterated on N' to improve results.

The approach is fast for 4-LUT mapping, as all optimum local subnetworks are precomputed. For k-LUT mappings, where $k > 4$, the exact synthesis may take a long time to find an optimum subnetwork by enumerating of all k-variable Boolean functions. In the context of LUT-based mapping that has thousands of LUTs, the execution time and quality of exact synthesis are both critical issues.

B. Improving Exact Synthesis by Logic Decomposition

The computational complexity of LUT-based exact synthesis is proportional to k. LUT mapping with larger k is of high interest as it increases the size of the subnetworks and therefore enables better optimization results.

As we stated in Section I, functional decomposition brings new opportunities for exact-synthesis-aware logic rewriting. Take the function shown in Fig. 2 as an example, instead of dealing with an n-variable function $f(x_1, \ldots, x_n)$ directly, the decomposed function $f = h(x_1, \ldots, x_{n-2}, g(x_{n-1}, x_n))$ has two subfunctions h and g, both of which have fewer inputs to leverage the computational complexity of exact synthesis. However, this kind of decomposition is effective only when the DSD or support-reducing decomposition exist. For the networks that cannot be disjointly decomposed, the decomposed logic network represented by XMG can serve as the starting point for exact synthesis to achieve incremental improvement, the process which is illustrated in Fig. 1(b).

The exact synthesis algorithm is shown in Algorithm 3, which consists of two parts. In lines 2–7, we start from a lower bound to check whether there exists an optimum network within a given timeout limit. If the timeout is exceeded, the first part stops and reports a timeout, we then try a heuristic approach in lines 8–15 to improve the upper bounds incrementally. The two functions are as follows.

1) `has_xmg`(f, r) returns an XMG, if the SAT solver checked that Boolean function f can be realized by a Boolean network of r gates.
2) `has_xmg_to`(f, r, t) acts the same as `has_xmg`(f, r), but terminates with no results after t seconds.

Note that different timeout strategies are used in the two parts. The timeout value is set to control the loop (lines 2–7) in the first part, while it controls each call of `has_xmg_to`(f, r, t) in the second one. Starting from $r = 0$, the former case behaves as if `has_xmg`(f, r) returns unsatisfiable and increments r by 1. Once r is large enough such that the problem is satisfiable, an optimum solution may be found within the time limit. In contrast, given r as an upper bound of XMG size, we decrease r by 1 to incrementally improve the XMG size if `has_xmg_to`(f, r, t) returns a satisfiable solution within t seconds. Concerning line 16 in Algorithm 2, we set the exact synthesis threshold $l = 4$, because the optimal XMGs of all 4-variable functions are precomputed.

Algorithm 3: Exact Synthesis

Input : A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$
Output: An mDSD structure

```
1  function exact_syn (f)
       // Start from a lower bound
2      r ← 0;
3      repeat
4          res = has_xmg(f, r);
5          if res ≠ nil then return res;
6          else r ← r + 1;
7      until timeout;
       // Start from an upper bound
8      res ← recursive_decomp (f);// Algorithm 2
9       r ← size_of(res) −1;
10     while true do
11         res_new ← has_xmg_to(f, r, t);
12         if res_new ≠ nil then
13             r ← r − 1; res ← res_new
14         else return res;
15     end
```

The complete flow of exact-synthesis-based decomposition is shown in Algorithm 4. The function f is first checked for its DSD-decomposability. If f can be written as the composition of subfunctions g and h, the decomposition process is recursively applied to g and h. Otherwise, the exact synthesis routine in Algorithm 3 is invoked for f.

VII. Experimental Results

We have evaluated the proposed functional decomposition 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.

A. Evaluation on DSD Benchmarks

We have implemented our logic decomposition approach in C++ on top of the logic synthesis framework CirKit.[1] The functions considered are partial-DSD or non-DSD functions that frequently occur in practical technology mapping applications [49]. Each function set contains 1000 functions. The name of the function set indicates the type and number of function inputs, e.g., "pdsd6" means 1000 partial-DSD decomposable 6-input functions, "ndsd8" means 1000 non-DSD decomposable 8-input functions. Our decomposition results are verified by simulating the truth tables of the resulting mDSD structure or XMG.

To exploit the decomposition capability of MAJ decomposition, in this experiment we first disable exact synthesis and Shannon decomposition in Algorithm 2. The experimental results are shown in Table I. Columns 2–7 give the numbers and percentages of functions that contain no prime nodes (FULL, isomorphic to an XMG), just one prime node (NONE, no decomposition exist), and the others which do not belong to

[1] github.com/msoeken/cirkit

Algorithm 4: Exact-Synthesis-Based Decomposition

Input : A Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$
Output: An mDSD structure

```
function xmg_by_dec (f)
    if is_DSD(f) then
        g, h ← Execute DSD of f;
        return xmg_by_dec (g); xmg_by_dec (h);
    end
    else
        return exact_syn (f); // Algorithm 3
    end
```

TABLE I
DECOMPOSITION RESULTS ON DSD BENCHMARKS

Func.	FULL	%(full)	PART	%(part)	NONE	%(none)	Time(s)
pdsd6	986	98.6	14	1.4	–	–	27
pdsd8	959	95.9	41	4.1	–	–	293
pdsd10	891	89.1	109	10.9	–	–	981
Average		**94.5**		**5.5**			
ndsd6	855	85.5	140	14	5	0.5	285
ndsd8	523	52.3	460	46	17	1.7	3176
ndsd10	440	44.0	511	51.1	49	4.9	7273
Average		**60.6**		**37**		**2.4**	

FULL and NONE (PART, the combination of prime nodes and basic gates). For partial-DSD functions, an averagely 94.5% functions can be decomposed into XMGs (FULL) and 5.5% functions are still partial-DSD (PART). In contrast, the non-DSD functions exhibit higher difficulty than the partial-DSD functions. After introducing MAJ decompositions, 60.6% of the non-DSD functions can be fully decomposed, 37% of the functions are transformed into partial-DSD, and only 2.4% functions still remain as non-DSD. As the number of input variables increases, the amount of fully decomposed functions also decreases. For example, our method can fully decompose 98.6% of the partial-DSD functions of the set dsd6 (6-input functions), whereas this number falls to 44% for the set of non-DSD functions dsd10 (10-input functions).

To evaluate the performance of MAJ decomposition on the XMG size and depth, we enable all decomposition types in Algorithm 2 and set $l = 4$ for exact synthesis. The experimental results are shown in Table II, where size and depth are the total amounts of nodes and depth of XMGs by logic decomposition. The baseline is obtained by evaluating the XMGs which are generated by logic decomposition method without MAJ decomposition. By introducing MAJ decomposition, it is shown that our method can improve the size and depth of XMGs by 14% and 8%, respectively. Concerning the CPU time, the decomposition using MAJ is much slower than the non-MAJ decomposition, this is because the MAJ top-down decomposition has a computational complexity of $O(n^3)$, while the SMT-based cofactors optimization is computationally expensive for some extreme cases. Hence, more CPU time is required to compute each function for the MAJ decomposition.

B. Evaluation on EPFL Benchmarks

We implemented the method described in Section VI to show the effectiveness of the exact synthesis-based optimization approaches to 6-input LUT (6-LUT) mapping. Although we can process the functions with up to 16 inputs, we choose 6-LUT mapping for demonstration. This is because the experiments conducted on k-LUT mapping ($k > 6$) show that a larger k is not always performing better due to the overlapping in the subject graph.

TABLE II
XMG DEPTH AND SIZE IMPROVEMENT BY MAJ DECOMPOSITION

Func.	Without MAJ			With MAJ			Improv.	
	Size	Depth	Time(s)	Size	Depth	Time(s)	Size	Depth
pdsd6	6,316	4,809	1	5,282	4,573	185	0.84	0.95
pdsd8	9,197	6,384	2	8,078	6,159	190	0.88	0.96
pdsd10	12,370	8,097	4	10,948	7,656	528	0.89	0.95
ndsd6	9,276	4,847	1	7,805	4,467	441	0.84	0.92
ndsd8	16,616	7,371	4	12,797	6,181	1387	0.77	0.84
ndsd10	22,532	8,598	9	20,707	7,730	3133	0.92	0.90
Average							**0.86**	**0.92**

We use the EPFL combinational benchmark suite[2] for a comparison with [12]. Both our method and the method in [12] start with the same input networks, containing only AND gates. We set $l = 6$, and the timeout value to 1 min in Algorithm 3. To perform k-LUT mapping, we use the ABC command `if -K 6`.

As shown in Table III, the first ten benchmarks are arithmetic circuits, while the remaining ten are random control circuits. XMG size or depth can be improved by 19 out of 20 benchmarks, except *Decoder*, in which we got an increase in both XMG size and depth. By computing the average size/depth product, our method improves by 10% and 30% versus [12] concerning EPFL arithmetic and random control benchmarks. We also compare the results after 6-LUT mapping. Generally, XMG size optimization advantage carries over also to LUT mapping improvements. The results show that both LUT size and depth can be improved. In total, our method achieves 10% and 26% reduction of LUT size/depth product versus [12] for EPFL arithmetic and random control benchmarks. However, as recent research pointed out [50], optimization of the size and depth of a logic network may not necessarily result in reduced LUT size and depth. The statement also holds for our experiments. For instance, *Sine* performs less well on XMG size but improves LUTs count. In contrast, *Multiplier* can be optimized in terms of XMG size and depth, whereas it results in an increment of LUT size with the same LUT depth.

The CPU time listed in Table III ranges from less-than-1 to 129545 s. The CPU time is related to the LUT size as each 6-LUT represents a 6-variable function for decomposition. During decomposition, the CPU time used for each function varies considerably. As an example, *Adder* is mapped into 192 6-LUTs but the total CPU time is less than 1 s. In contrast, *Memory controller* is mapped into 11558 6-LUTs, while the total CPU time is up to 129545 s, which means 11 s are required on average to process a single function. The CPU time variation can be explained as follows according to Algorithm 4.

[2]lsi.epfl.ch/benchmarks

TABLE III
USING EXACT SYNTHESIS AND LOGIC DECOMPOSITION FOR XMG-SIZE OPTIMIZATION

Benchmarks	I/O	Previous method [12]				Our Method				
		XMG		6-LUT		XMG		6-LUT		
		Size	Depth	LUT Size	LUT Depth	Size	Depth	LUT Size	LUT Depth	Time(s)
Adder	256/129	383	129	192	64	383	**128**	192	64	< 1
Barrel Shifter	135/128	2,858	17	512	4	**2,315**	**14**	512	4	5,403
Divisor	128/128	39,768	4,310	13036	1097	**36,912**	**4,241**	**11,207**	**862**	163
Hypotenuse	256/128	99,927	9,017	44657	4455	**99,375**	**8,750**	**44,590**	**4,268**	7,357
Log2	32/32	23,006	219	7736	84	**22,699**	**185**	**7,556**	**78**	18,014
Max	512/130	1,982	254	744	90	**1,938**	**193**	765	**55**	37
Multiplier	128/128	16,575	136	5388	64	**16,388**	**133**	5,502	64	7,439
Sine	24/25	3,825	121	1460	42	3,877	**112**	**1,396**	**39**	8,025
Square-root	128/64	17,369	6,149	6161	1115	**17,184**	**5,052**	**6,064**	**1,029**	1,321
Square	64/128	8,527	155	3846	63	**8,300**	**151**	**3,819**	**61**	4,065
Average Size · Depth			43,930,095.4		5,926,551.0		39,694,647.9		5,323,779.7	
Improvement (new/old)							**0.90**		**0.90**	
Round-robin arbiter	256/129	12,159	89	3,819	20	**12,138**	89	**2,722**	**18**	< 1
Alu control unit	7/26	118	8	29	2	**107**	**7**	29	2	592
Coding-cavlc	10/11	715	17	139	4	**703**	18	**135**	4	16,078
Decoder	8/256	304	3	287	2	312	4	**272**	2	< 1
i2c controller	147/142	1,287	18	379	5	**1,269**	**17**	**348**	5	8,014
Int to float converter	11/7	247	15	56	4	**224**	17	**49**	4	4,191
Memory controller	1204/1231	42,580	125	12,727	35	**39,308**	**105**	**11,558**	**31**	129,545
Priority encoder	128/8	753	245	238	31	**690**	**115**	**219**	**27**	1
Lookahead XY router	60/30	211	46	91	11	**190**	**31**	**90**	**5**	12
Voter	1001/1	6,838	60	2,429	16	**6,728**	**58**	**2,023**	**14**	1,598
Average Size · Depth			408,227.1		26,252.2		284,294.1		19,538.4	
Improvement (new/old)							**0.70**		**0.74**	

6-LUT indicates the LUT size and depth after 6-LUT technology mapping using ABC command `if -K 6`

TABLE IV
RESULTS ON ISCAS'85 BENCHMARKS COMPARED TO AIG REWRITING

Benchmarks	I/O	AIG Rewriting				Our Method				
		AIG Size	AIG Depth	LUT Size	LUT Depth	XMG Size	XMG Depth	LUT Size	LUT Depth	Time(s)
c432	36/7	129	25	48	8	131	25	49	8	< 1
c499	41/32	387	17	78	4	181	13	**65**	4	14
c880	60/26	314	21	83	6	239	19	**74**	6	43
c1355	41/32	390	18	78	4	188	14	**60**	4	6
c1908	33/25	360	25	86	6	246	22	**81**	6	347
c2670	233/140	572	17	134	5	376	19	198	5	1,458
c3540	50/22	927	31	257	8	846	29	**246**	8	3,645
c5315	178/123	1,297	26	298	6	893	22	**279**	7	6,406
c6288	32/32	1,870	88	521	16	1,096	46	**512**	17	2,622
c7552	207/108	1,407	24	403	6	999	28	**379**	7	3,077
Average		765.3	29.2	198.6	6.9	**519.5**	**23.7**	**194.3**	7.2	

1) If a function is full-DSD or partial-DSD with a prime function that has a small number of input variables, then the solution can be returned immediately.
2) If a function is non-DSD, our exact synthesis algorithm first starts from a lower bound to find an optimum representation for this function. If the timeout happens after 1 min, we call the functional decomposition to obtain an XMG, which is an upper bound for the exact synthesis. During the process of incremental improvement, the CPU time is positively related to how much size improvement can be achieved as we check the satisfiability for each improvement step within 1 min.

C. Evaluation on ISCAS Benchmarks

To compare the XMG-based synthesis method with AIG rewriting for size optimization, we applied the AIG optimization script "`resyn; resyn2`" iteratively to the ISCAS benchmark suite in ABC until convergence. Table IV shows the results of both the size and depth of AIGs and XMGs. After optimization, we report the LUT size and depth using k-LUT mapping. The LUT size can be reduced for 8 out of 10 benchmarks, except *c432* and *c2670*. The most significant improvement is *c1355*, while the LUT size is reduced from 78 by AIG rewriting to 60 by our method. In terms of LUT depth, the last three benchmarks result in a small increase. The CPU time listed in the last column of Table IV shows that our method obtained better results at the cost of more computational efforts.

D. Evaluation on Quantum Reciprocal Operation

In [7], a synthesis algorithm called direct XMG synthesis (DXS) has been proposed to realize quantum circuits based on XMGs. The general idea is to map each gate in an XMG into a small quantum circuit and then compose these circuits. However, quantum computers are limited to perform reversible computations, which requires them to store intermediate results on auxiliary qubits. Besides the number of qubits, the cost of a quantum circuit is measured in terms of the number of T gates. The T gate accounts for the far most complex execution in a quantum computer [51]. We show how

TABLE V
RESULTS ON QUANTUM CIRCUITS REALIZATION OF RECIPROCAL OPERATION (INTDIV(n))

n	Previous method [7]				Our method (% improvement)				
	Normal		Bennett		Normal (%)		Bennett (%)		
	#qubits	#T gates	#qubits	#T gates	#qubits	#T gates	#qubits	#T gates	Time(s)
16	1,109	10,976	494	15,526	9.4	11.9	11.7	10.4	2,549
32	4,590	44,380	1,929	62,538	6.0	8.7	10.6	6.1	1,305
64	18,090	169,988	7,337	239,386	2.5	3.8	7.0	1.1	2,989
128	70,712	650,916	28,056	916,692	1.4	2.3	5.3	-0.3	3,304
Average					**4.8**	**6.7**	**8.7**	**4.3**	

the improved XMGs affect the quality of the network by applying DXS to XMGs obtained from [12] and from the proposed method. As benchmarks we used the integer reciprocal design INTDIV(n) for $n = 16, 32, 64, 128$ [7]. Table V lists the results, where "Qb" means the number of qubits and "*T*g" means the number of T gates. DXS comes in two variants: *Normal* and *Bennett*. The latter typically leads to fewer qubits for the sake of a higher number of T gates. It can be seen that due to the more compact XMGs both the qubits and number of T gates improve (except for the 128-bit version, for which the amount of T gates slightly increases).

VIII. CONCLUSION

In this paper, we proposed a logic decomposition algorithm using majority operator (MAJ) and utilized it in several applications. Given a Boolean function represented by a truth table, we first exploit decomposition properties using majority both from top-down and bottom-up decomposition scenarios. Then, we propose an algorithm that combines the decomposition using MAJ with conventional DSD decomposition to derive XMGs. The algorithm is applied to exact-synthesis-aware rewriting and quantum circuit synthesis. Compared to state-of-the-art algorithms based on XMGs over EPFL benchmark suites, the proposed algorithm enables a 10% (26%) size/depth product reduction for arithmetic (random control) benchmarks in 6-LUT circuits mapped by ABC.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for useful comments.

REFERENCES

[1] M. M. Waldrop, "The chips are down for Moore's law," *Nat. News*, vol. 530, no. 7589, pp. 144–147, 2016.
[2] L. Amarú, P.-E. Gaillardon, S. Mitra, and G. De Micheli, "New logic synthesis as nanotechnology enabler," *Proc. IEEE*, vol. 103, no. 11, pp. 2168–2195, Nov. 2015.
[3] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition," in *Proc. DAC*, 2013, pp. 1–6.
[4] I. Amlani, A. O. Orlov, G. Toth, G. H. Bernstein, C. S. Lent, and G. L. Snider, "Digital logic gate using quantum-dot cellular automata," *Science*, vol. 284, no. 5412, pp. 289–291, 1999.
[5] T. Schneider, A. A. Serga, B. Leven, B. Hillebrands, R. L. Stamps, and M. P. Kostylev, "Realization of spin-wave logic gates," *Appl. Phys. Lett.*, vol. 92, no. 2, 2008, Art. no. 022505.
[6] A. Imre, G. Csaba, L. Ji, A. Orlov, G. H. Bernstein, and W. Porod, "Majority logic gate for magnetic quantum-dot cellular automata," *Science*, vol. 311, no. 5758, pp. 205–208, 2006.
[7] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *Proc. DATE*, 2017, pp. 470–475.
[8] C.-C. Tsai and M. Marek-Sadowska, "Multilevel logic synthesis for arithmetic functions," in *Proc. DAC*, 1996, pp. 242–247.
[9] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. DAC*, 2014, pp. 1–6.
[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. DAC*, 2006, pp. 532–535.
[11] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.
[12] W. Haaswijk, M. Soeken, L. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *Proc. ASPDAC*, 2017, pp. 151–156.
[13] E. A. Ernst, "Optimal combinational multi-level logic synthesis," Ph.D. dissertation, Dept. Comput. Sci. Eng., Univ. Michigan, Ann Arbor, MI, USA, 2009.
[14] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *Proc. ICCD*, 2011, pp. 429–430.
[15] W. J. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "LUT mapping and optimization for majority-inverter graphs," in *Proc. IWLS*, 2016, pp. 1–8.
[16] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," in *Proc. DATE*, 2016, pp. 1030–1035.
[17] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Proc. DATE*, 2017, pp. 830–835.
[18] W. J. Haaswijk, A. Mischenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," in *Proc. DAC*, 2018, Art. no. 53.
[19] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," in *Proc. DATE*, 2000, pp. 208–215.
[20] Z. Chu, M. Soeken, Y. Xia, and G. De Micheli, "Functional decomposition using majority," in *Proc. ASPDAC*, 2018, pp. 676–681.
[21] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *Proc. ICCAD*, 1997, pp. 78–82.
[22] R. L. Ashenhurst, *The Decomposition of Switching Functions*, vol. 29, Comput. Lab, Harvard Univ., Cambridge, MA, USA, pp. 74–116, 1959.
[23] H. A. Curtis, *A New Approach to the Design of Switching Circuits*. New York, NY, USA: van Nostrand, 1962.
[24] R. M. Karp, "Functional decomposition and switching circuit design," *J. Soc. Ind. Appl. Math.*, vol. 11, no. 2, pp. 291–335, 1963.
[25] A. Mishchenko, "Enumeration of irredundant circuit structures," in *Proc. IWLS*, 2014, pp. 1–7.
[26] A. Mishchenko, R. K. Brayton, and S. Chatterjee, "Boolean factoring and decomposition of logic networks," in *Proc. ICCAD*, 2008, pp. 38–44.
[27] A. Mishchenko, S. Cho, S. Chatterjee, and R. K. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007, pp. 354–361.
[28] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.
[29] V. Bertacco, S. Minato, P. Verplaetse, L. Benini, and G. De Micheli, "Decision diagrams and pass transistor logic synthesis," in *Proc. IWLS*, 1997, pp. 1–12.
[30] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 7, pp. 866–876, Jul. 2002.

[31] S.-I. Minato and G. De Micheli, "Finding all simple disjunctive decompositions using irredundant sum-of-products forms," in *Proc. ICCAD*, 1998, pp. 111–117.
[32] A. Mishchenko and R. Brayton, "Faster logic manipulation for large designs," in *Proc. IWLS*, 2013, pp. 1–6.
[33] V. Callegaro, F. S. Marranghello, M. G. A. Martins, R. P. Ribas, and A. I. Reis, "Bottom-up disjoint-support decomposition based on cofactor and Boolean difference analysis," in *Proc. ICCD*, 2015, pp. 680–687.
[34] Y. Tohma, "Decompositions of logical functions using majority decision elements," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 6, pp. 698–705, Dec. 1964.
[35] S. B. Akers, "Synthesis of combinational logic using three-input majority gates," in *Proc. 3rd Annu. Symp. Switch. Circuit Theory Logical Design*, 1962, pp. 149–158.
[36] V. N. Kravets and K. A. Sakallah, "Resynthesis of multi-level circuits under tight constraints using symbolic optimization," in *Proc. ICCAD*, 2002, pp. 687–693.
[37] M. Soeken, P. Raiola, B. Sterin, B. Becker, G. De Micheli, and M. Sauer, "SAT-based combinational and sequential dependency computation," in *Proc. Haifa Verification Conf.*, 2016, pp. 1–17.
[38] S. R. Das, P. K. Srimani, and C. R. Datta, "On multiple fault analysis in combinational circuits by means of Boolean difference," *Proc. IEEE*, vol. 64, no. 9, pp. 1447–1449, Sep. 1976.
[39] R. A. Smith, "Minimal three-variable NOR and NAND logic circuits," *IEEE Trans. Electron. Comput.*, vol. EC-14, no. 1, pp. 79–81, Feb. 1965.
[40] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*, 2010, pp. 24–40.
[41] M. Soeken, L. G. Amaru, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 11, pp. 1842–1855, Nov. 2017.
[42] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Proc. Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, 2009, pp. 32–44.
[43] N. Eén, "Practical SAT—A tutorial on applied satisfiability solving," in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, 2007.
[44] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Boston, MA, USA: Addison-Wesley, 2015.
[45] E. Goto and H. Takahasi, "Some theorems useful in threshold logic for enumerating Boolean functions," in *Proc. Int. Feder. Inf. Process. Congr.*, 1962, pp. 747–752.
[46] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli, "Classifying functions with exact synthesis," in *Proc. ISMVL*, 2017, pp. 272–277.
[47] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New Delhi, India: McGraw-Hill Higher Educ., 1994.
[48] K. L. Kodandapani and S. C. Seth, "On combinational networks with restricted fan-out," *IEEE Trans. Comput.*, vol. C-27, no. 4, pp. 309–318, Apr. 1978.
[49] A. Mishchenko, "An approach to disjoint-support decomposition of logic functions," Dept. Elect. Comput. Eng., Portland State Univ., Portland, OR, USA, Rep., 2001. [Online]. Available: https://people.eecs.berkeley.edu/∼alanmi/publications/2001/tech01_dsd.pdf
[50] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for LUT-based technology mapping," in *Proc. FPGA*, 2017, pp. 147–156.
[51] S. Bravyi and A. Kitaev, "Universal quantum computation with ideal Clifford gates and noisy ancillas," *Phys. Rev. A*, vol. 71, no. 2, 2005, Art. no. 022316.

Zhufei Chu (M'14) received the B.S. degree in electronic engineering from Shandong University, Weihai, China, in 2008, and the M.S. and Ph.D. degrees in communication and information system from Ningbo University, Ningbo, China, in 2011 and 2014, respectively.

He was a Post-Doctoral Fellow with the École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland, from 2016 to 2017. He is currently an Associate Professor with Ningbo University. His current research interests include many aspects of logic synthesis and its applications.

Mathias Soeken (M'13) received the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2013.

He is a Scientist with the École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland. He is investigating constraint-based techniques in logic synthesis and industrial-strength design automation for quantum computing. He is actively maintaining the logic synthesis frameworks CirKit and RevKit. His current research interests include many aspects of logic synthesis and formal verification.

Dr. Soeken was a recipient of the scholarship from the German Academic Scholarship Foundation. He has been serving as a TPC Member for several conferences, including DAC, DATE, and ICCAD. He is a Reviewer for *Mathematical Reviews* as well as for several journals.

Yinshui Xia received the B.S. degree in physics and the M.S. degree in electronic engineering from Hangzhou University, Zhejiang, China, in 1984 and 1991, respectively, and the Ph.D. degree in electronic engineering from Edinburgh Napier University, Edinburgh, U.K., in 2003.

He was a Visiting Scholar with Kings College London, London, U.K., in 1999 and then joined Edinburgh Napier University as a Research Assistant and an Enterprise Fellow from 2000 to 2005. He is currently a Professor with Ningbo University, Ningbo, China. His current research interests include low-power digital circuit design, logic synthesis and optimization, and SoC design.

Lunyao Wang received the Ph.D. degree in circuits and systems from Zhejiang University, Hangzhou, China, in 2012.

He is currently a Professor with Ningbo University, Ningbo, China. His current research interests include power and area optimization in CMOS circuit design and CMOS/nanowire/molecular hybrid cell assignment. His current research interests include Reed–Muller functions synthesis and optimization and approximate logic synthesis for error tolerant applications.

Giovanni De Micheli (F'94) received the Nuclear Engineering degree from the Politecnico di Milano, Milan, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, in 1980 and 1983, respectively.

He is a Professor and the Director of the Institute of Electrical Engineering, École Polytechnique Fédéderale de Lausanne, Lausanne, Switzerland. He has authored the book entitled *Synthesis and Optimization of Digital Circuits* (McGraw-Hill, 1994), and has coauthored and/or coedited 9 other books and over 800 technical articles. He has Google Scholar citations with an H-index of 94. He is a member of the Scientific Advisory Board of IMEC, Leuven, Belgium, CfAED, Dresden, Germany, and STMicroelectronics, Geneva, Switzerland. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies, networks on chips, bio-sensros/systems, and data processing of biomedical information.

Prof. De Micheli was a recipient of the 2016 IEEE/CS Harry Goode Award for seminal contributions to design and design tools of networks-on-chips, the 2016 EDAA Lifetime Achievement Award, the 2012 IEEE/CAS Mac Van Valkenburg Award for contributions to theory, practice, and experimentation in design methods and tools, the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems, the D. Pederson Award for the Best Paper on the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS in 1987 and 2018, and several best paper awards, including DAC in 1983 and 1993, DATE in 2005, Nanoarch in 2010 and 2012, and Mobihealth in 2016. He has been serving IEEE in several capacities, namely, the Division 1 Director in 2008 and 2009, the Co-Founder and the President Elect of the IEEE Council on EDA from 2005 to 2007, the President of the IEEE CAS Society in 2003, the Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 1997 to 2001. He has been the chair of several conferences, including Memocode in 2014, DATE in 2010, pHealth in 2006, VLSI SOC in 2006, DAC in 2000, and ICCD in 1989. He is a fellow of ACM and a member of the Academia Europaea and an International Honorary Member of the American Academy of Arts and Sciences.