Functional Decomposition Using Majority

Zhufei Chu[1,2], Mathias Soeken[2], Yinshui Xia[1], and Giovanni De Micheli[2]
[1]Faculty of Electrical Engineering & Computer Science, Ningbo University, Ningbo, China
[2]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

Abstract— Typical operators for the decomposition of Boolean functions in state-of-the-art algorithms are AND, exclusive-OR (XOR), and a 2-to-1 multiplexer (MUX). We propose a logic decomposition algorithm that uses the majority-of-three (MAJ) operation. Such decomposition can extend the capabilities of current logic decomposition, but only found limited attention in previous work. Our algorithm makes 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 (i) exact synthesis aware rewriting, (ii) pre-optimization for 6-LUT mapping, and (iii) synthesis of quantum networks. An experimental evaluation shows that our algorithm leads to better XMGs compared to state-of-the-art algorithms, which positively affect all these three applications. As one example, our experiments show that the proposed method achieves up to 37.1% with a average of 9.6% reduction on the look-up tables (LUT) size/depth product applied to the EPFL arithmetic benchmarks after technology mapping.

I. Introduction

Boolean decomposition is the task of representing a (complex) Boolean function in terms of a basis consisting of simple subfunctions. *Disjoint-support decomposition* (DSD) is a special case of Boolean decomposition, which results in a tree of nodes with non-overlapping supports and inverters as optional complemented attributes on the edges. Due to its low implementation cost, DSD received wide interest and is applied to designing both ASIC and FPGA.

Typical decomposition operators are AND, XOR, and a 2-to-1 multiplexer (MUX). Algorithms using AND/OR/MUX representations are generally exploited to synthesize control logic (AND/OR-intensive). In contrast, datapath logic, especially when arithmetic functions are involved, makes extensive use of XOR and the majority-of-three function $\langle xyz \rangle = xy \vee xz \vee yz$ (MAJ, [1]). Moreover, nanotechnologies such as Quantum-Dot Cellular Automata [2], Spin Wave Devices [3], and Nanomagnets [4], realize majority gates as primitive building blocks. Also, in commonly used cost models for quantum computing, MAJ can be implemented at the same cost of AND/OR, and the cost of an XOR gate can be neglected [5]. Therefore, it is advantageous that logic synthesis methods unify AND/OR and XOR/MAJ representations to support different circuit designs.

Logic representations that use MAJ as a basic logic primitive have recently been proposed for the synthesis of Boolean logic functions. The MAJ-based logic representation demonstrated superior synthesis results for both standard CMOS and emerging technologies [6]. For instance, *XOR-majority graphs* (XMGs, [7]) have XOR, MAJ, and inverters as logic primitives, which are an extension of the *Majority-inverter graphs* (MIGs) introduced in [8].

XMGs offer a compact logic representation when being used for *exact synthesis*, since it allows for compact networks in which each of the gate has a small number of fanins [7]. Exact synthesis algorithms have been proposed that find an optimum XMG in terms of size or depth for a given Boolean function. Due to its high computational complexity, exact synthesis is generally limited to small functions. However, recent studies show how the combination of exact synthesis and logic rewriting led to improvements in *And-Inverter Graphs* (AIGs) [9, 10], MIG [11, 6], and XMG size optimization [7]. Logic decomposition can strengthen exact synthesis, because (i) large network can be decomposed into small subnetworks using DSD or support-reducing decomposition techniques [12]; (ii) the upper bound of optimization objective can be computed by logic decomposition, which can be provided as the start point for Satisfiability (SAT) solver to enable incremental improvement.

The early attempts to achieve MAJ logic decomposition in 60's were concerned with the existence of MAJ decomposition based on truth tables or Karnaugh map [13, 14]. Due to their intractable complexity, failed to gain interest later in automated logic synthesis. Known recent decomposition algorithms that yield MAJ operation are mostly based on *binary decision diagrams* (BDDs) [1, 12, 15]. A constructive library-aware multilevel logic synthesis approach using symmetries is proposed in [12], in which MAJ cell is used as one primitive library, the method integrates the technology-independent and technology-dependent stages of synthesis to favor ultimate topological and physical structures. The synthesis flow is later applied in the resynthesis loop under tight industry constraints [15]. Another work try to decompose a function F as $\langle F_a F_b F_c \rangle$ by constructive method. Even the method searching majority dominator nodes on BDDs, the solution space is still huge and need high-effort computation to construct MAJ. The key difference of our work is that (i) functional decomposition using MAJ with one disjoint-support is considered instead of library-aware decomposition, and (ii) truth-tables and mDSD structure are used to manipulate operations instead of BDD.

In this paper, we extend the capability of current logic decomposition methods by additionally using MAJ. Our contributions are as follows:

1. We make use of a MAJ decomposition which resembles the well-known Shannon decomposition: we show that under some conditions it is possible to write a function F as $\langle xGH \rangle$ such that G and H do not depend on variable x (Section III).
2. We propose a decomposition algorithm that combines DSD, MAJ, and Shannon decomposition to factorize an XMG from a function provided as truth table (Section IV).
3. We improve exact synthesis aware logic rewriting, in optimum or near-optimal XMGs are derived for each LUT (lookup-table) in a LUT-network (Section V).

Experimental results on Boolean functions reveal that using

MAJ enables more logic decomposition opportunities. On average, 87.3% partial-DSD functions (functions which are partially decomposable by DSD) can be decomposed into XMGs; while 54.7% functions allowing no DSD decomposition whatsoever (non-DSD) can be decomposed into XMGs. The proposed logic decomposition algorithm is integrated into an exact synthesis algorithm by decomposing large logic network into small subnetwork and computing the upper bounds of optimization objective. Experimental results show that the proposed method achieves up to 38.1% with a average 8.6% reduction on XMG size/depth product, while up to 37.1% with a average of 9.6% reduction on the look-up tables (LUT) size/depth product, applied to the EPFL arithmetic benchmarks.

II. Background

Boolean Functions Given a set of Boolean variables $X = \{x_1, \dots, x_n\}$, 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, e.g., [16]). 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 $F(x_1, \dots, x_i, \dots, x_n)$ wrt. variable x_i is $F_{x_i} = F(x_1, \dots, 1, \dots, x_n)$, and the negative cofactor is $F_{\bar{x}_i} = F(x_1, \dots, 0, \dots, x_n)$.

The basic Boolean operations considered in this paper are AND, OR, XOR, and MAJ. For the purposes of distinction, we refer to all basic operations except MAJ as *ordinary operations*. MAJ can be expressed in disjunctive, conjunctive normal form, and exclusive-or-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

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

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

Logic Representations Typically, multi-level logic networks are represent as directed acyclic graphs, called dags, in which terminal nodes are input variables or constants and internal nodes are logic operations. Homogeneous logic networks, which restrict the nodes' functions to be from a small set of functions, have attracted more interest due to its simplicity and thus optimization opportunities. Popular instances of homogeneous logic representations include NAND and NOR circuits [17], *AND-inverter graphs* (AIGs) [18], and recently proposed MIGs [8]. XMGs are an extension of MIGs which additionally use XOR.

A *DSD structure* is a data structure to manage the computational operations of decomposition/composition functions, which was first introduced by Mishchenko [19]. In this paper, to obtain an XMG, we use a modified DSD structure, referred to as *mDSD structure*, to manage the computational operations. An mDSD structure over the primary input variables $X = \{x_1, \dots, x_n\}$ is a dag $T = (V, E, Y)$ with

- a finite set of nodes $V = X \cup G$, where $G = \{g_1, \dots, g_k\}$ are internal nodes representing the logic operations in the tree,

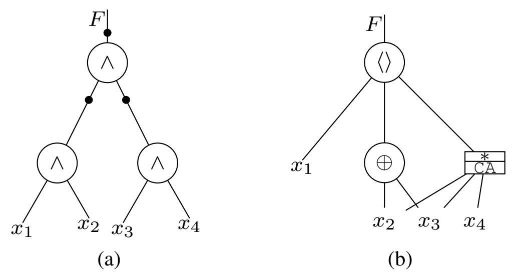

Fig. 1. Example mDSD structure representations (a) $F = x_1x_2 \vee x_3x_4$ where edges with bubble indicate the complemented attribute (b) $F = \langle x_1(x_2 \oplus x_3)\texttt{PRIME}(x_2, x_3, x_4)\rangle$.

- 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,
- and a finite multiset of outputs $Y \in V \times \mathbb{B}$.

For each internal node $g_i \in G$, the operations can be basic gates (AND, OR, XOR, MAJ) or *prime* nodes. Each prime node is associated with a hexadecimal truth table, which indicates it needs further functional decomposition. The truth table of a prime node is called prime function if it cannot be decomposed. 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. We do not consider MUX disjoint decomposition in order to obtain a data-structure that is closer to the XMG representation.

If the resulting mDSD structure has no prime nodes, it is isomorphic to an XMG, which can be directly derived from it. Consequently, the proposed logic decomposition algorithm enables further opportunities in the synthesis with XMG networks.

Fig. 1 shows examples of two mDSD structures. Note that OR can be derived from AND using De Morgan's rule. Function $F = x_1x_2 \vee x_3x_4$ in Fig. 1 (a) contains three ANDs and three inverters, that is $F = \overline{\overline{x_1x_2} \wedge \overline{x_3x_4}}$. The function $F = \langle x_1(x_2 \oplus x_3)\texttt{PRIME}(x_2, x_3, x_4)\rangle$ in Fig. 1 (b) contains a prime node, denoted by '$\star$', with truth table `0xCA`.

Disjoint-Support Decomposition The decomposition of a logic function $F(x_1, \dots, x_n)$ identifies a set of functions $A_i(x_i)$ with no shared input variables, and a function L [20] such that:

$$F = L(A_0, \dots, A_i, \dots)$$

The DSD is a special case of Boolean logic decomposition. Function $F(X)$ has a disjoint decomposition when the two other functions, say G and H, such that:

$$F(x_1, \dots, x_n) = H(x_1, \dots, x_{i-1}, G(x_i, \dots, x_n))$$

$$F(X) = H(X_1, G(X_2)),\ X_1 \cup X_2 = X \ \ X_1 \cap X_2 = \emptyset \tag{3}$$

In contrast, F is said to be prime function if it cannot be represented by (3).

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.

Exact Synthesis Exact synthesis is the task of finding an optimum logic network representation for a given input specification in terms of size or depth [21]. Using size as an optimization objective, as an example, the idea is to use a SAT solver to check whether there exists a Boolean network of r gates that realizes the given functions [22]. The algorithm solves a sequence of decision problems. The algorithm starts with $r = 1$ and incrementally increases r until the SAT solver returns a satisfiable solution. That means the search for a size-optimum network with r gates requires to solve up to r decision problems using a SAT solver.

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 immediately return the solution. 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 6 variables), it can also be implemented for large functions when being applied to small subnetworks to guarantee local optimality [22].

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. As an example, 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 up to 200,253,952,527,185 NPN classes for $n = 6$ [23].

III. Majority Logic Decomposition

Our work makes use of functional decomposition based on the majority operation. The aim is to represent F as $\langle xGH\rangle$ such that G and H do not depend on x. We make use of the following decompostion [14]:

Theorem 1. *Let F be a Boolean function and x a variable in F. Then*

$$F = \langle xF_xF_{\bar{x}}\rangle \qquad \textit{if, and only if } F \textit{ is monotone.} \tag{4}$$

Proof. If F is monotone, then $\overline{F}_xF_{\bar{x}} = 0$. From Shannon's decomposition (in XOR-form), we know that $F = xF_x \oplus \bar{x}F_{\bar{x}}$. Also we have $\langle xF_xF_{\bar{x}}\rangle = xF_x \oplus xF_{\bar{x}} \oplus F_xF_{\bar{x}}$ (see Eq. (1)). We check for which condition these two equations differ:

$$\begin{aligned} & xF_x \oplus \bar{x}F_{\bar{x}} \oplus xF_x \oplus xF_{\bar{x}} \oplus F_xF_{\bar{x}} \\ \overset{a\oplus a=0}{=}\; & \bar{x}F_{\bar{x}} \oplus xF_{\bar{x}} \oplus F_xF_{\bar{x}} \\ \overset{\bar{a}b\oplus ab=b}{=}\; & F_{\bar{x}} \oplus F_xF_{\bar{x}} \overset{a\oplus ab=a\bar{b}}{=} \overline{F}_xF_{\bar{x}}. \end{aligned}$$

Hence the equations only differ whenever $\overline{F}_xF_{\bar{x}} = 1$. Because F is monotone, $\overline{F}_x$ and $F_{\bar{x}}$ have an empty intersection. Therefore, we can write $F = \langle xF_xF_{\bar{x}}\rangle$. □

IV. Functional Decomposition using Majority

This section describes our proposed functional decomposition algorithm. Input to the algorithm is a Boolean function $F : \mathbb{B}^n \to \mathbb{B}$ represented as truth table. The output is an XMG.

```
1  function recursive_decomp(F)
2      R ←mDSD_init(F);
3      repeat
4          R ←func_decomp(R);
5      until exist_no_prime_node(R);
6      return R;
7  function func_decomp(R)
8      for each prime node n in R do
9          f ←truth table of n;
10         if is_basic(f) then update_basic(n,R);
11         else if is_DSD(f) then update_DSD(n,R);
12         else if inputs_of_(f) ≤ m then update_exact(n,R);
13         else if is_MAJ(f)∧ flag then update_MAJ(n,R);
14         else update_Shannon(n,R);
15     end
16     return R;
```

Algorithm 1: Functional decomposition

A. Algorithm Overview

Given a truth table, the algorithm uses exhaustive evaluation for all prime nodes until it cannot be decomposed. The computational results are operated by the mDSD structure. The algorithm is outlined in Algorithm 1. After checking special cases and initialization, it constructs an mDSD structure with just one prime node. The decomposability of the prime node is checked according to following conditions in a sequential way.

(i) If the truth table equals the basic functions such as two-inputs AND, OR, XOR, and three-inputs MAJ, then the prime node will be replaced by these basic gates (line 10).

(ii) For each input of the prime node, we traverse all input variables to check whether it can be DSD-decomposed (line 11).

(iii) If no DSD exists, the prime node must be a prime function, if the prime function having up to m variables, we call exact synthesis subroutine described in [21] (line 12).

(iv) If the three previous conditions do not hold, we check whether Eq.(4) is satisfied to try the proposed MAJ decomposition. Meanwhile, we set a flag to check whether the MAJ decomposition wrt. input variable achieves best support size of cofactors (See Section IV. B for details) (line 13). Otherwise, the Shannon decomposition is applied (line 14).

Note that any of these conditions is satisfied, we create a corresponding operator node and new prime node with updated truth table and supports. Finally, the resulting mDSD structure is returned as the solution. Since Shannon decomposition is implemented, the mDSD structure is isomorphic to an XMG.

The conditions for MAJ decompositions are demonstrated in Section III. For the conditions and updating schemes of DSD, readers can refer to [20, 24] for further details.

B. Heuristic to Select MAJ or Shannon

For prime functions that are not DSD-decomposable, the variables that satisfy Eq.(4) may not be unique. Different variables result in distinct cofactors. Suppose both $x_i \in X$ and $x_j \in X$, $i \neq j$, satisfy Eq.(4), then

$$F = \langle x_iF_{x_i}F_{\bar{x}_i}\rangle = \langle x_jF_{x_j}F_{\bar{x}_j}\rangle \tag{5}$$

The support size of distinct cofactors, which is the key issue to control the subsequent size and depth of mDSD structure,

```
1  function exact_synthesis($F$)
2      $r \leftarrow 1$;
3      repeat
4          $res = \texttt{has\_xmg}(F, r)$;
5          if $res \neq nil$ then return res;
6          else $r \leftarrow r + 1$;
7      until timeout;
8      $res \leftarrow recursive\_decomp(F)$; // Algorithm 1
9      $r \leftarrow$ size_of($res$) $-1$;
10     while true do
11         $res\_new \leftarrow \texttt{has\_xmg\_to}(F, r, t)$;
12         if $res\_new \neq nil$ then
13             $r \leftarrow r - 1$; $res \leftarrow res\_new$
14         else return $res$;
15     end
```

Algorithm 2: Exact Synthesis

also may not be equal. Therefore, instead of using MAJ decomposition as a priority to decompose prime functions, we first calculate the minimum support size of cofactors, referred to as y_s, of all variables in X. Then, we exploit the variables that satisfy Eq.(4), similarly, and calculate the minimum support size of cofactors, referred to as y_m, among these validated variables. The MAJ decomposition is implemented only when $y_m \leq y_s$ by setting `flag=true` in line 12 of Algorithm 1. Otherwise, Shannon decomposition is applied.

V. 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-LUTs. It provides an attractive way to identify the subnetworks. Given an input network N, the approach proposed in [7] 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 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 together 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 have thousands of LUTs, the execution time and quality of exact synthesis are both important 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 subnetwork and enables better optimization results. To leverage the computational complexity, we propose a logic decomposition method to improve exact synthesis performance. The idea is based on two principles: (i) large network can be decomposed into the combinations of disjoint-supports and prime functions with fewer variables; (ii) for networks that cannot be disjointly decomposed, the upper bounds of the XMG can be computed by Algorithm 1, which is served as the starting point for exact synthesis to achieve incremental improvement.

Given a k-inputs function obtained from the LUT, we first call Algorithm 1 with following settings:

- MAJ- and Shannon-based decomposition are disabled. We defer such kind of decompositions as we first try DSD for each function to obtain optimal small subnetworks in size.
- We set $m = k$ in Algorithm 1 line 12.

For functions that are not DSD-decomposable, Algorithm 2 is invoked to return a solution and all decompostion types are enabled.

There are two parts in Algorithm 2. In lines 2–7, we start from a lower bound to check whether there exists an optimum network within given timeout value constraint. If the timeout exceeds, the first part stops and reports a timeout, we then try heuristic in lines 8–15 to improve the upper bounds incrementally. The two functions,

- $\texttt{has\_xmg}(F, r)$ returns an XMG if the SAT solver checked Boolean function F can be realized by a Boolean network of r gates;
- $\texttt{has\_xmg\_to}(F, r, t)$ acts the same with $\texttt{has\_xmg}(F, r)$, but terminates with no results after t seconds.

Timeout Note that the different timeout strategies are used in the two parts. The timeout value is set to control the loop (lines 2–7) in terms of the first part, while control each call of $\texttt{has\_xmg\_to}(F, r, t)$ in the second one. Starting from $r = 1$, the former case behaves as if $\texttt{has\_xmg}(F, r)$ returns *unsatisfiable* and increments r by 1. Once r is large enough that the problem is *satisfiable*, an optimum solution may be found within the time limit. In contrast, given r an upper bound of XMG size, we decrease r by 1 to incremental improve the XMG size if $\texttt{has\_xmg}(F, r, t)$ returns satisfiable solution within t seconds.

Computing Upper Bound Given a Boolean function F, Algorithm 1 will return an XMG quickly if we set the exact synthesis threshold as $m = 4$. To this end, all decomposition types are used sequentially to obtain an XMG. It should be pointed out that F must be a prime function in this context, as exact synthesis are only applied to prime functions. During logic decomposition of F, the first step would be exact synthesis, MAJ, or Shannon decomposition, then the subsequent decomposition steps could be any of decomposition types. The worst case is the XMG used as the upper bound be returned as a solution, if no more improvements can be achieved by decreasing r.

VI. Experimental Results

We evaluate 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.

TABLE I
DECOMPOSITION RESULTS ON DSD BENCHMARKS

Sets	Inputs	#Func.	FULL	%(full)	PART	%(part)	NONE	%(none)
Partial	6	1M	983,017	98.3	16,983	1.7	–	–
	8	1M	954,870	95.5	45,130	4.5	–	–
	10	100K	92,392	92.4	7,608	7.6	–	–
	12	100K	88,645	88.6	11,355	11.4	–	–
	14	10K	7,581	75.8	2,419	24.2	–	–
	16	10K	7,294	72.9	2,706	27.1	–	–
Avg.				**87.3**		**12.7**		
None	6	1M	929,116	92.9	59,423	5.9	11,461	1.1
	8	1M	692,390	69.2	264,884	26.5	42,726	4.3
	10	100K	49,731	49.7	42,490	42.5	7,779	7.8
	12	100K	47,478	47.5	47,810	47.8	4,712	4.7
	14	10K	3,192	31.9	6,084	60.8	724	7.2
	16	10K	3,710	37.1	5,990	59.9	300	3.0
Avg.				**54.7**		**40.6**		**4.7**

A. Evaluation on DSD benchmarks

We implemented our logic decomposition approach in C++ on top of the logic synthesis framework CirKit.[1] The DSD benchmarks[2] considered are enumerated partial-DSD and non-DSD functions with 6–16 inputs, which are not fully DSD-decomposable using only AND, OR, and XOR. Our decomposition results are verified by simulating the truth tables of the resulting mDSD structure or XMG.

To exploit the decomposition capability of MAJ, here we first disable exact synthesis and Shannon decomposition in Algorithm 1. The experimental results are shown in Table I, which includes two benchmark sets. As '#Func.' indicates, there are 1 million 6-input functions, while 10 thousands 16-input functions, et al. The last six columns 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 does not belong to FULL and NONE (PART, the combination of prime nodes and basic gates). On average, 87.3% functions that are partially DSD decomposable can be decomposed into XMGs (FULL). Further, 54.7% functions exhibiting no possibility for DSD decomposition can be decomposed into XMGs (FULL), and 40.6% be decomposed into PART, after introducing MAJ decomposition.

As the number of input variables increased, the percentage of FULL functions are generally decreased. For partial-DSD functions set, up to 98.3% functions of 1 million 6-inputs benchmark are FULL by our method. The minimum percentage is the 16-inputs benchmark, which achieves 72.9% FULL functions. There are only 4.7% functions are still not decomposable (NONE).

To evaluate the performance of MAJ decomposition on the XMG size and depth, we enable all decomposition types in Algorithm 1 and set $m = 4$ for exact synthesis. The experimental results on non-DSD benchmarks are shown in Table II, where size and depth are the total amount of gates 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. It is shown that our method can improve size and depth by 5.4% and 5.9%, respectively.

[1]`github.com/msoeken/cirkit`
[2]`people.eecs.berkeley.edu/~alanmi/temp5`

TABLE II
XMG DEPTH AND SIZE IMPROVEMENT BY MAJ DECOMPOSITION

Set	Inputs	#Func.	Base		Improvement (%)	
			Size	Depth	Size	Depth
None	6	1M	7,924,723	4,583,829	4.4	3.2
	8	1M	13,661,959	6,989,645	7.3	6.2
	10	100K	1,962,317	839,971	5.1	5.0
	12	100K	2,906,682	1,023,392	7.4	6.8
	14	10K	408,819	130,521	4.0	6.9
	16	10K	610,317	148,447	4.5	7.0
Avg.					**5.4**	**5.9**

B. Evaluation on EPFL benchmarks

We implemented the method described in Section V to show the effectiveness of the exact synthesis based optimization approaches to 6-LUT mapping. The EPFL arithmetic benchmark[3] are considered for comparison with [7]. Both our method and [7] start with the same input networks, containing only AND gates. We set $m = 6$, and timeout value to one minute in Algorithm 2. In terms of k-LUT mapping, we focus on default technology mapping using ABC command `if -K 6`.

As shown by Table III, XMG size or depth can be improved by 9 out of 10 benchmarks, except *Sine*, in which we got increase in both XMG size and depth. By computing the geometric mean, taken over the size and depth of the networks, our method performs 4.8% reduction of geomean, and 8.6% reduction of size/depth product than [7]. We also compare the results after 6-LUT mapping. Generally, XMG size optimization advantage also carries over into LUT mapping improvements. The results show that both LUT size and depth can be improved. In total, our method achieves 5.8% reduction of geomean of LUT size and depth, and 9.6% reduction of LUT size/depth product than [7]. However, as a recent research pointed out [25], optimization of the size and depth of a logic network may not necessarily result in reduced LUT size and depth. The statement is also hold for our experiment. For instance, *Sine* performs less well on both XMG size and depth, but achieves improvement of LUT size and depth. In contrast, *Multiplier* can be optimized in terms of XMG size and depth, whereas results in increasement of LUT size with exactly the same LUT depth.

C. Evaluation on Quantum Reciprocal Operation

In [5] a synthesis algorithm called DXS (direct XMG synthesis) has been proposed to realize quantum networks based on an XMGs. The general idea is to map each gate in an XMG into a quantum network and then compose these networks. However, quantum computers are limited to perform reversible computations which requires to store intermediate results on auxiliary qubits. Besides the number of qubits, the cost of a quantum network is measured in terms of the number of T gates. The T gate accounts for the far most complex execution in a quantum computer [26]. We show how the improved XMGs affect the quality of the network by applying DXS to XMGs obtained from [7] and from the proposed method. As benchmarks we used the integer reciprocal design `INTDIV`(n) for $n = 16, 32, 64, 128$ [5]. Table IV lists the results, where 'Qb' means number of qubits and 'Tg' means 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

[3]`lsi.epfl.ch/benchmarks`

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

Benchmarks	I/O	Previous method [7]				Our Method			
		XMG		6-LUT		XMG		6-LUT	
		Size	Depth	LUT Size	LUT Depth	Size	Depth	LUT Size	LUT Depth
Adder	256/129	383	129	192	64	383	128	192	64
Barrel Shifter	135/128	2858	17	512	4	2149	14	512	4
Divisor	128/128	39768	4310	13036	1097	37003	4243	11189	862
Hypotenuse	256/128	99927	9017	44657	4455	99428	8755	44615	4293
Log2	32/32	23006	219	7736	84	22957	213	7573	78
Max	512/130	1982	254	744	90	1938	200	766	55
Multiplier	128/128	16575	136	5388	64	16357	133	5554	64
Sine	24/25	3825	121	1460	42	3896	140	1400	39
Square-root	128/64	17369	6149	6161	1115	17187	5169	6234	1028
Square	64/128	8527	155	3846	63	8325	156	3813	61
Avg.		21,422	2,050.7	8,373.2	707.8	**20,962.3**	**1,915.1**	**8,184.8**	**654.8**
GeoMean		1,807.5		633.9		**1,721.6**		**597.4**	
Size · Depth		43,930,095.4		5,926,551		**40,144,900.7**		**5,359,407**	

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

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

n	Previous method [5]					Our method (% improvement)				
	time(s)	Normal		Bennett		time(s)	Normal (%)		Bennett (%)	
		Qb	Tg	Qb	Tg		Qb	Tg	Qb	Tg
16	0.82	1,109	10,976	494	15,526	643.16	9.4	11.4	13.2	8.9
32	1.42	4,590	44,380	1,929	62,538	233.30	6.1	8.5	11.6	5.6
64	2.07	18,090	169,988	7,337	239,386	649.59	2.3	3.4	7.2	0.5
128	2.68	70,712	650,916	28,056	916,692	763.29	1.3	2.1	4.9	-0.5
Avg.							**4.8**	**6.3**	**9.2**	**3.6**

number of T gates. Note that we did not count the CPU time of precomputed optimum XMG library construction in [5]. Hence, the proposed approach requires more runtime as indicated in the Table, which includes both the time to compute the XMG and the time used by DXS, which is negligible. However, it can be seen that due to the more compact XMGs both the qubits and number of T gates improves (except for the 128-bit version, for which the number of T gates slightly decreases).

VII. CONCLUSION

In this paper we make use of a decomposition based on MAJ and proposed an algorithm that combines the decomposition with conventional DSD decomposition in order to derive XMGs. The introducing of MAJ decomposition extends the capability of logic decomposition. The 54.7% functions exhibiting no possibility for DSD decomposition can be decomposed into XMGs. The logic decomposition method is applied to exact synthesis aware rewriting and quantum network synthesis. Experimental results show the effectiveness of our method on both XMG/LUT size and depth optimization, and the number of qubits and T gates optimization in quantum network.

Acknowledgments This research was partly supported by H2020-ERC-2014-ADG 669354 CyberCare, by the Swiss National Science Foundation (200021-169084 MAJesty), by NSF China 61501268, by Zhejiang Provincial NSF LQ15F040001, and by Ningbo City NSF 2015A610112.

REFERENCES

[1] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition," in *DAC*, p. 47, ACM, 2013.
[2] 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.
[3] T. Schneider, A. Serga, B. Leven, B. Hillebrands, R. Stamps, and M. Kostylev, "Realization of spin-wave logic gates," *Applied Physics Letters*, vol. 92, no. 2, p. 022505, 2008.
[4] A. Imre, G. Csaba, L. Ji, A. Orlov, G. Bernstein, and W. Porod, "Majority logic gate for magnetic quantum-dot cellular automata," *Science*, vol. 311, no. 5758, pp. 205–208, 2006.
[5] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *DATE*, pp. 470–475, 2017.
[6] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," in *DATE*, pp. 1030–1035, 2016.
[7] W. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *ASPDAC*, pp. 151–156, 2017.
[8] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *DAC*, pp. 1–6, 2014.
[9] N. Li and E. Dubrova, "AIG rewriting using 5-input cuts," in *ICCD*, pp. 429–430, 2011.
[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Design Automation Conference*, pp. 532–535, 2006.
[11] W. J. Haaswijk, M. Soeken, L. Amaru, P.-E. Gaillardon, and G. De Micheli, "LUT mapping and optimization for majority-inverter graphs," in *IWLS*, 2016.
[12] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," in *DATE*, pp. 208–215, 2000.
[13] Y. Tohma, "Decompositions of logical functions using majority decision elements," *IEEE Transactions on Electronic Computers*, no. 6, pp. 698–705, 1964.
[14] S. B. Akers, "Synthesis of combinational logic using three-input majority gates," in *Third Annual Symposium on Switching Circuit Theory and Logical Design*, pp. 149–158, 1962.
[15] V. N. Kravets and K. A. Sakallah, "Resynthesis of multi-level circuits under tight constraints using symbolic optimization," in *ICCAD*, pp. 687–693, 2002.
[16] M. Soeken, P. Raiola, B. Sterin, B. Becker, G. De Micheli, and M. Sauer, "SAT-based combinational and sequential dependency computation," in *Haifa Verification Conference*, pp. 1–17, 2016.
[17] R. A. Smith, "Minimal three-variable NOR and NAND logic circuits," *IEEE Transactions on Electronic Computers*, no. 1, pp. 79–81, 1965.
[18] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *CAV*, pp. 24–40, 2010.
[19] A. Mishchenko and R. Brayton, "Faster logic manipulation for large designs," in *IWLS*, Citeseer, 2013.
[20] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *ICCAD*, pp. 78–82, 1997.
[21] M. Soeken, L. G. Amaru, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.
[22] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *DATE*, pp. 830–835, 2017.
[23] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli, "Classifying functions with exact synthesis," in *ISMVL*, pp. 272–277, 2017.
[24] V. Callegaro, F. S. Marranghello, M. G. Martins, R. P. Ribas, and A. I. Reis, "Bottom-up disjoint-support decomposition based on cofactor and boolean difference analysis," in *ICCD*, pp. 680–687, IEEE, 2015.
[25] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for lut-based technology mapping," in *FPGA*, pp. 147–156, 2017.
[26] S. Bravyi and A. Kitaev, "Universal quantum computation with ideal Clifford gates and noisy ancillas," *Physical Review A*, vol. 71, no. 2, p. 022316, 2005.