This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2789725, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



# Stochastic Circuit Synthesis by Cube Assignment

Xuesong Peng and Weikang Qian, *Member, IEEE*

***Abstract*—Stochastic computing (SC) is an unconventional computation paradigm, in which digital circuits are adopted to compute on stochastic bit streams. The value represented by a stochastic bit stream is the probability of obtaining a one in the stream. Stochastic circuits are highly tolerant to bit flip errors. Compared to the conventional binary computing, SC can perform complicated arithmetic computations with simple circuits. With such advantages, SC has been applied in a number of applications. This raises recent interests in developing general methods to automatically synthesize stochastic circuits. However, the synthesis problem is different from and more complicated than the traditional logic synthesis, due to the special solution space of the problem. In this work, we propose a novel method to synthesize a high-quality stochastic circuit. Our method is based on assigning cubes (i.e., product terms) to the on-set of the Boolean function. A heuristic breadth-first search algorithm is proposed to search for a good stochastic circuit in the solution space. Our experimental results showed that the proposed method can produce better circuits than the state-of-the-art methods.**

***Index Terms*—stochastic computing, stochastic circuit synthesis, cube assignment, general stochastic circuits, logic synthesis.**

## I. Introduction

As semiconductor industry enters into the nano-scale regime, reliability has become a paramount concern. Stochastic computing (SC) [1], an unconventional computing paradigm, has attracted attention due to its strong tolerance to bit flip errors. In SC, a real-valued number $x \in [0, 1]$ is represented by a stochastic bit stream, in which every bit has probability $x$ of being a one and probability $1 - x$ of being a zero. As an example, the stream $A$ in Fig. 1 contains four 1's out of total eight bits, so it encodes the value $0.5$.

Since SC uses a uniform-weighted encoding, which usually needs a long bit stream to represent a value, a single bit flip in the stream does not change the value significantly. As a result, SC is highly tolerant to bit flip errors [2].

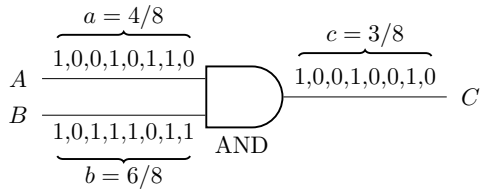


Fig. 1: An AND gate performs multiplication on real values encoded by stochastic bit streams.

Under the SC paradigm, many arithmetic functions can be implemented by very simple circuits. Fig. 1 shows an example where multiplication is realized by a single AND gate. Assume that the two input stochastic bit streams are independent. Then, the probability of a one in the output stream of the AND is $P(C = 1) = P(A = 1) \cdot P(B = 1)$, where $P(X = 1)$ represents the probability of a one in the bit stream $X$. Given its low hardware cost, SC has been used in several applications, such as image processing [3], [4], [5], digital filters [6], [7], decoding of modern error-correcting codes [8], [9], and artificial neural networks [10], [11], [12].

Early stochastic circuits were designed manually. Several basic computing units, such as multiplier, scaled adder, divider, and squaring unit, were proposed in this way [13]. However, they can only perform limited types of computations. In order to implement more arithmetic functions with SC, researchers recently proposed several systematic synthesis methods [2], [14], [15], [16], [17], [18], [19]. However, one specific challenge in stochastic circuit synthesis is its extremely large design space: there are many different Boolean functions realizing the same target function [16], [19]. Since different Boolean functions have different costs, a critical problem is to find a valid Boolean function with the smallest hardware cost. One common feature of most previous methods is that they only construct one specific solution for the design target. Although these methods are very fast, they do not pay effort to search the solution space for a better solution. As a result, the synthesized circuit may still be far from the optimal solution. The only exception is the work [19]. However, it can only implement a special type of polynomial called *multi-linear polynomial*.

In this work, we propose a novel method that searches the space of the valid Boolean functions to derive a better combinational stochastic circuit. Our method is general enough to implement any polynomial as long as it maps the unit interval into the unit interval, a necessary condition for a polynomial to be realized by a combinational stochastic circuit [20]. The basic idea is to iteratively add *cubes* (i.e., *product terms*) into the on-set of a Boolean function. The set of cubes added must satisfy specific constraints so that the final Boolean function realizes the target function in SC. Nevertheless, at each iteration, there exist many valid cubes that can be selected. In order to synthesize a good solution, we propose a heuristic breadth-first search algorithm to explore the solution space.

In summary, the main contributions of this work are as follows.

- We propose a new method that iteratively selects cubes to construct a Boolean function for a target computation in SC.
- We develop a heuristic breadth-first search algorithm to search for a good solution.
- We propose a basic version of the algorithm that works for any univariate function and an extended version that works for any multivariate polynomials.

A preliminary version of this work was published in [21], in which a branch-and-bound-based algorithm is used to explore the solution space. Compared to that version, we have three major improvements in this work. We introduce a breadth-first search algorithm, which is much more runtime-efficient than the previous algorithm. We extend our method to handle arbitrary multivariate polynomials. We also modify the algorithm so that it can synthesize a high-quality multi-level circuit, even though the primary optimization target is a two-level circuit.

The rest of the paper is organized as follows. In Section II, we discuss the related works. In Section III, we give the background on a general design of stochastic circuit, which is the focus of this work. We also present the key optimization problem in synthesizing stochastic circuits of that form. In Section IV, we present our proposed solution based on cube assignment. In Section V, we present extensions to handle

Xuesong Peng and Weikang Qian are with the University of Michigan - Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China, 200240. Email: xuesongp@126.com; qianwk@sjtu.edu.cn.

multivariate polynomials. In Section VI, we analyze the time and space complexity of the proposed method. In Section VII, we show the experimental results. Finally, in Section VIII, we conclude the paper.

## II. Related Works

Several previous works have proposed methods to synthesize stochastic circuits. These methods can be classified into two types. The first type of methods synthesize reconfigurable stochastic circuits [20], [22], [23], [14], [24]. These circuits can be configured to implement different functions. In [20], the authors proposed to synthesize a combinational circuit to realize univariate polynomial computation. Their method first transforms the target polynomial into a Bernstein polynomial and then realizes it by a circuit consisting of an adder and a multiplexer. The work [22] extended the method in [20] to handle multivariate polynomials. In [23], [14], and [24], the authors proposed methods to implement SC by sequential circuits. Their methods model a sequential circuit with stochastic input bit streams as a Markov chain and exploit the steady state probability distribution to synthesize the target computation.

The second type of methods synthesize fixed stochastic circuits [25], [15], [19], [16], [17], [18], [26]. The synthesized circuit can implement only one specific function. Compared to reconfigurable stochastic circuits, fixed circuits take less area. The works [17], [26], [19] proposed methods for special type of target functions. In [17], the authors proposed a method to synthesize stochastic circuits for arithmetic functions. Their method implements the Maclaurin series expansion of a target function. It first factorizes the polynomial and then realizes each factor by a series of NAND gates. However, the restrictions of this method include that the coefficients of a factor should be alternately positive and negative and that their magnitudes should be monotonically decreasing. In [26], the authors proposed a double-NAND structure for realizing polynomial functions. However, the method can only synthesize polynomials with positive coefficients. In [19], the authors showed that different Boolean functions could compute the same arithmetic function in SC. They introduced the concept of stochastic equivalence class and proposed a method to search for the optimal Boolean function within an equivalence class. However, their method can only synthesize multi-linear polynomials.

The works [18], [25], [15], [16] proposed methods to synthesize fixed stochastic circuits for general polynomials. It should be noted that our work also falls into this category. The method in [18] is based on factorizing the polynomial into first-order and second-order factors and implementing each factor by a stochastic circuit. However, for some second-order factors, a stochastic implementation requires a costly stochastic subtractor. In [25], the authors established a fundamental relation between stochastic circuits and spectral transform. They proposed a general method to synthesize stochastic circuits through this relation. The work [15] further extends the work [25] by proposing a method to optimize the sub-circuit that provides stochastic bit streams of constant probabilities, which are needed in the stochastic implementation. In [16], the authors designed a general combinational stochastic circuit to realize arithmetic computation. Their work also described a method to synthesize a low-cost stochastic circuit. However, as we mentioned in Section I, these methods directly construct a solution within the solution space without paying effort to search for a better solution.

## III. Background on Synthesizing Stochastic Circuits

Our proposed method is based on a general form of stochastic circuit proposed in [16]. In this section, we introduce this general form and discuss the key optimization problem in synthesizing a target function. In what follows, when we say the probability of a signal, we mean the probability of the signal to be a one.

### A. General Stochastic Circuit and Its Computation

A general form of a stochastic circuit is shown in Fig. 2. It is a combinational circuit. The inputs of the circuit can be partitioned into $k+1$ sets. The inputs in the $i$-th $(1 \leq i \leq k)$ set are $X_{i,1}, \ldots, X_{i,d_i}$. The last set of inputs are $Y_1, \ldots, Y_m$. The stochastic bit streams fed into these inputs are independent. For each $1 \leq i \leq k$, the probabilities of the bit streams to inputs $X_{i,1}, \ldots, X_{i,d_i}$ are all set as a variable probability $0 \leq x_i \leq 1$. The inputs $Y_1$, ..., $Y_m$ are all supplied with stochastic bit streams with constant probabilities $0.5$. As will be shown shortly, these input streams of $0.5$ probability are used to realize different constant coefficients of a target polynomial. Some previous works [2], [17], [26] assume the availability of arbitrary constant probabilities. However, different from conventional binary computing, in SC, the generation of constant probabilities requires additional circuits. Typically, a constant probability generator includes *unbiased random bit sources* that generate $0.5$ probabilities (e.g., the DFFs in a linear feedback shift register (LFSR) [27]) and an additional circuit that converts the $0.5$ probabilities into the final probability (e.g., a comparator [2]). In this sense, $0.5$ probabilities are the ultimate constant input probabilities and hence, we assume the inputs $Y_i$'s are provided with these $0.5$ probabilities. Furthermore, this choice eliminates the boundary between the stochastic datapath and the constant probability generators and hence, allows a larger optimization space. The number of these inputs of $0.5$ probability, $m$, affects the quantization error and is chosen according to the precision requirement on the coefficients of the target polynomial. For example, if the coefficients of the polynomial should have a precision of $1/2^8$, then $m$ is chosen as $8$. The larger the value $m$ is, the smaller the quantization error will be. Therefore, we refer to $m$ as the *precision parameter*.

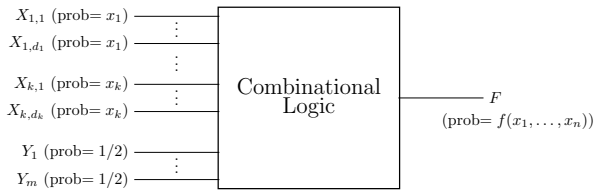


Fig. 2: A general form of a stochastic circuit.

Suppose the Boolean function of the combinational circuit in Fig. 2 is $B(X_{1,1}, \ldots, X_{1,d_1}, \ldots, X_{k,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m)$. Now, we analyze the function realized by the stochastic circuit, which is encoded by the output stochastic bit stream.

For simplicity, we define $n = \sum_{i=1}^{k} d_i$. First, we introduce the following two definitions.

**Definition 1.** *For any $0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$, we define the set*

$$M(s_1, \ldots, s_k) = \{(a_{1,1}, \ldots, a_{k,d_k}, b_1, \ldots, b_m) \in \{0,1\}^{n+m} : \sum_{j=1}^{d_i} a_{i,j} = s_i, \text{ for all } i = 1, \ldots, k\}. \quad \square$$

**Definition 2.** *For a Boolean function $B$, its* on-set*, denoted as $On(B)$, is the set of input vectors that let the Boolean function evaluate to* 1. $\square$

Consider any input vector $(a_{1,1}, \ldots, a_{k,d_k}, b_1, \ldots, b_m) \in \{0,1\}^{n+m}$. Suppose it is in a set $M(s_1, \ldots, s_k)$. The probability that the random input vector $(X_{1,1}, \ldots X_{k,d_k}, Y_1, \ldots, Y_m) = (a_{1,1}, \ldots, a_{k,d_k}, b_1, \ldots, b_m)$ is

$$\prod_{i=1}^{k} \prod_{j=1}^{d_i} Pr(X_{i,j} = a_{i,j}) \cdot \prod_{i=1}^{m} P(Y_i = b_i). \quad (1)$$

By the probability configuration, for any $1 \leq i \leq k$ and $1 \leq j \leq d_i$, $Pr(X_{i,j} = 1) = x_i$ and $Pr(X_{i,j} = 0) = 1 - x_i$; for any $1 \leq i \leq m$, $Pr(Y_i = 1) = Pr(Y_i = 0) = 0.5$. Furthermore, since $\sum_{j=1}^{d_i} a_{i,j} = s_i$, the probability value shown in Eq. (1) is equal to

$$\frac{1}{2^m} \prod_{i=1}^{k} x_i^{s_i} (1 - x_i)^{d_i - s_i}.$$

The above analysis indicates that for any two input vectors in the same set $M(s_1, \ldots, s_k)$, the probability of a random input vector equal to the first is same as the probability of a random input vector equal to the second. In contrast, for any two input vectors in two different sets $M(s_1, \ldots, s_k)$, the probability of a random input vector equal to the first is different from the probability of a random input vector equal to the second. Therefore, the sets $M(s_1, \ldots, s_k)$ form a set of equivalence classes over $\{0,1\}^{n+m}$. The cardinality of the set $M(s_1, \ldots, s_k)$ is $2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$.

The output function $f(x_1, \ldots, x_k)$ of the stochastic circuit is the probability that $B(X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m) = 1$, given that the input vector $(X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m)$ is random. We have

$$\begin{aligned} &f(x_1, \ldots, x_k) = Pr(B(X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m) = 1) \\ &= \sum_{\substack{(a_{1,1}, \ldots, a_{k,d_k}, \\ b_1, \ldots b_m) \in On(B)}} \frac{P((X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m) = (a_{1,1}, \ldots, a_{k,d_k}, b_1, \ldots, b_m))}{}. \end{aligned}$$

Since we can partition the on-set $On(B)$ into subsets $On(B) \cap M(0, \ldots, 0)$, …, $On(B) \cap M(d_1, \ldots, d_k)$, we can represent $f(x_1, \ldots, x_k)$ as

$$\begin{aligned} &f(x_1, \ldots, x_k) \\ &= \sum_{s_1=0}^{d_1} \cdots \sum_{s_k=0}^{d_k} \sum_{\substack{(a_{1,1}, \ldots, a_{k,d_k}, \\ b_1, \ldots b_m) \in \\ On(B) \cap M(s_1, \ldots, s_k)}} \frac{P((X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m) = (a_{1,1}, \ldots, a_{k,d_k}, b_1, \ldots, b_m))}{} \\ &= \sum_{s_1=0}^{d_1} \cdots \sum_{s_k=0}^{d_k} \frac{|On(B) \cap M(s_1, \ldots, s_k)|}{2^m} \prod_{i=1}^{k} x_i^{s_i} (1 - x_i)^{d_i - s_i}, \end{aligned} \tag{2}$$

where $|S|$ denotes the cardinality of the set $S$.

**Example 1.** *Consider a combinational circuit with $k = 2$, $d_1 = 2$, $d_2 = 1$, and $m = 2$. Suppose its Boolean function is $B(X_{1,1}, X_{1,2}, X_{2,1}, Y_1, Y_2) = X_{1,1}\overline{X_{1,2}}\,\overline{Y_1}Y_2 + X_{2,1}Y_1$. By definition,*

$$\begin{aligned} M(1,1) &= \{(a_{1,1}, a_{1,2}, a_{2,1}, b_1, b_2) \in \{0,1\}^5 : \\ &\qquad a_{1,1} + a_{1,2} = 1, a_{2,1} = 1\} \\ &= \{(1,0,1,b_1,b_2), (0,1,1,b_1,b_2) : (b_1, b_2) \in \{0,1\}^2\}. \end{aligned}$$

*Therefore, we have*

$$\begin{aligned} O(B) \cap M(1,1) = \{&(1,0,1,0,1), (1,0,1,1,0), (1,0,1,1,1), \\ &(0,1,1,1,0), (0,1,1,1,1)\}. \end{aligned}$$

*and $|O(B) \cap M(1,1)| = 5$. Similarly, we can obtain*

$$\begin{aligned} &|O(B) \cap M(0,0)| = 0, |O(B) \cap M(0,1)| = 2, \\ &|O(B) \cap M(1,0)| = 1, |O(B) \cap M(2,0)| = 0, \\ &|O(B) \cap M(2,1)| = 2. \end{aligned}$$

*Therefore, the function realized by the stochastic circuit is*

$$\begin{aligned} f(x_1, x_2) = &\frac{1}{2}(1 - x_1)^2 x_2 + \frac{1}{4} x_1 (1 - x_1)(1 - x_2) \\ &+ \frac{5}{4} x_1 (1 - x_1) x_2 + \frac{1}{2} x_1^2 x_2. \quad \square \end{aligned}$$

### *B. Synthesis of Arbitrary Target Function and the Key Optimization Problem*

Given a target arithmetic function on $k$ variables $x_1, \ldots, x_k$, we first approximate it as a multivariate polynomial $f(x_1, \ldots, x_k)$. As long as the multivariate polynomial evaluates inside the interval $(0,1)$ for all $0 \leq x_1 \leq 1$, …, $0 \leq x_k \leq 1$, we can apply a method proposed in [22] to transform it into a multivariate Bernstein polynomial [28] with all the coefficients in the unit interval of the following form

$$\begin{aligned} &f_1(x_1, \ldots, x_k) \\ &= \sum_{s_1=0}^{d_1} \cdots \sum_{s_k=0}^{d_k} \alpha_{s_1 \ldots s_k} \prod_{i=1}^{k} \binom{d_i}{s_i} x_i^{s_i} (1 - x_i)^{d_i - s_i}, \end{aligned} \tag{3}$$

where $0 \leq \alpha_{s_1 \ldots s_k} \leq 1$ ($0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$) are the constant coefficients. As we will show shortly, our method can realize any multivariate Bernstein polynomial with all coefficients in the unit interval. Therefore, our method can synthesize any multivariate polynomial that evaluates inside $(0,1)$ for all $0 \leq x_1 \leq 1$, …, $0 \leq x_k \leq 1$. Note that since SC encodes values by probabilities, a polynomial that can be realized by a stochastic circuit must be in the interval $[0,1]$ for all $0 \leq x_1 \leq 1$, …, $0 \leq x_k \leq 1$. Therefore, our method can synthesize almost all polynomials that can be realized by SC theoretically. For a polynomial that evaluates outside the interval $(0,1)$ for some $0 \leq x_1 \leq 1, \ldots, 0 \leq x_k \leq 1$, we cannot implement it by our method. To synthesize it, we need to first perform an affine transformation to make it fall into the valid range.

In what follows, we suppose that by applying the techniques mentioned above, the original function is transformed into a multivariate Bernstein polynomial with all coefficients in the unit interval as shown in Eq. (3). For any $0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$, we let $G(s_1, \ldots, s_k)$ be an integer rounded from the value

$$\alpha_{s_1 \ldots s_k} \cdot 2^m \cdot \prod_{i=1}^{k} \binom{d_i}{s_i}.$$

Then, we have

$$\frac{G(s_1, \ldots, s_k)}{2^m} \approx \alpha_{s_1 \ldots s_k} \prod_{i=1}^{k} \binom{d_i}{s_i}.$$

and our target function is further approximated as

$$\begin{aligned} &f_2(x_1, \ldots, x_k) \\ &= \sum_{s_1=0}^{d_1} \cdots \sum_{s_k=0}^{d_k} \frac{G(s_1, \ldots, s_k)}{2^m} \prod_{i=1}^{k} x_i^{s_i} (1 - x_i)^{d_i - s_i}. \end{aligned} \tag{4}$$

Note that since $\alpha_{s_1 \ldots s_k}$ is in the unit interval, $G(s_1, \ldots, s_k)$ is an integer in the range $[0, 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}]$.

In summary, by applying the above sequence of transformations, the original target function is transformed into the one shown in Eq. (4). Comparing Eq. (4) with Eq. (2), we find that any function $B$ satisfying that for all $0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$,

$$|On(B) \cap M(s_1, \ldots, s_k)| = G(s_1, \ldots, s_k)$$

will realize the given target function. In other words, if there are $G(s_1, \ldots, s_k)$ vectors in the set $M(s_1, \ldots, s_k)$ that make

the Boolean function $B$ evaluate to 1, then the Boolean function $B$ realizes the target function in SC. However, there are a large number of Boolean functions satisfying the requirement. If we want to synthesize an optimal circuit, we need to find an optimal Boolean function that satisfies the requirement. In our work, we primarily focus on two-level circuits, and we use the literal count in the sum-of-product (SOP) form as the cost measure [29]. However, it is also possible to synthesize a multi-level circuit from the obtained optimal SOP, which is demonstrated to have a good quality by our experimental results.

In summary, the optimization problem we consider is as follows:

> Given an integer $m$ and $\prod_{i=1}^{k}(1+d_i)$ integers $G(0,\ldots,0)$, $\ldots$, $G(d_1,\ldots,d_k)$ such that $0 \le G(s_1,\ldots,s_k) \le 2^m \prod_{i=1}^{k}\binom{d_i}{s_i}$ for all $0 \le s_1 \le d_1$, $\ldots$, $0 \le s_k \le d_k$, determine a Boolean function $B(X_{1,1},\ldots,X_{1,d_1},\ldots,X_{k,1},\ldots,X_{k,d_k},Y_1,\ldots,Y_m)$ with minimal literal count satisfying that for all $0 \le s_1 \le d_1,\ldots,0 \le s_k \le d_k$,
> $$|On(B) \cap M(s_1,\ldots,s_k)| = G(s_1,\ldots,s_k).$$

To facilitate the solving of the optimization problem, we represent a Boolean function in the solution space, $B(X_{1,1}, \ldots, X_{k,d_k}, Y_1, \ldots, Y_m)$, as a matrix, where the columns represent the $X$-variables and the rows represent the $Y$-variables. As an example, Fig. 3 shows such a matrix for the Boolean function $B(X_{1,1},X_{1,2},X_{1,3},Y_1,Y_2) = \overline{X_{1,1}}\,\overline{Y_1} + X_{1,2}\overline{Y_1} + \overline{X_{1,1}}X_{1,3}$. For this case, $k=1$, $d_1=3$, and $m=2$.

| $Y\backslash X$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | 1 | 1 | 1 | 1 | | | 1 | 1 |
| 01 | 1 | 1 | 1 | 1 | | | 1 | 1 |
| 10 | | 1 | | 1 | | | | |
| 11 | | 1 | | 1 | | | | |

Fig. 3: The matrix representation of the Boolean function $B(X_{1,1},X_{1,2},X_{1,3},Y_1,Y_2) = \overline{X_{1,1}}\,\overline{Y_1} + X_{1,2}\overline{Y_1} + \overline{X_{1,1}}X_{1,3}$.

To understand the problem under the matrix representation, we first introduce the following set definition.

**Definition 3.** *For any $0 \le s_1 \le d_1,\ldots,0 \le s_k \le d_k$, we define the set*

$$I(s_1,\ldots,s_k) = \{(a_{1,1},\ldots,a_{k,d_k}) \in \{0,1\}^n : \sum_{j=1}^{d_i} a_{i,j} = s_i, \text{ for all } i = 1,\ldots,k\}. \quad \square$$

With the matrix representation, the set $M(s_1,\ldots,s_k)$ is composed of all the columns with their indices in the set $I(s_1,\ldots,s_k)$. For simplicity, we say a column is in the set $I(s_1,\ldots,s_k)$ if its index is in the set $I(s_1,\ldots,s_k)$. The optimization problem is to distribute $G(s_1,\ldots,s_n)$ ones to the columns in the set $I(s_1,\ldots,s_n)$, for all $0 \le s_1 \le d_1,\ldots,0 \le s_k \le d_k$, to obtain an optimal Boolean function.

A method was proposed in the previous work [16] to find a solution. It applies a simple heuristic strategy to distribute the ones. However, this method only constructs a single solution without paying any effort to explore the solution space. Thus, the solution may be further improved. In our work, we propose a search-based method to find a better solution.

Note that although the optimization problem has flexibility in determining the final Boolean function, it is different from the traditional logic minimization with *don't cares* or Boolean relation minimization problem [30]. The problem we consider here specifies the number of input vectors of a set that can be assigned into the on-set of the Boolean function. However, neither logic minimization with *don't cares* nor Boolean relation minimization can constrain the number of input vectors of a set that can be assigned into the on-set. Therefore, new methods are needed to solve the optimization problem.

## IV. Proposed Method by Cube Assignment

In this section, we present our proposed method by cube assignment. For simplicity, we first explain how it works for univariate polynomials, i.e., the cases where $k=1$. We will show the extensions to handle multivariate polynomials in Section V. For univariate cases, the number $n = d_1$, and the $d_1$ $X$-inputs $X_{1,1}$, $X_{1,2}$, $\ldots$, $X_{1,d_1}$ are simply denoted as $X_1$, $\ldots$, $X_n$.

### A. Preliminaries

In this section, we first introduce some notations and definitions which will be used later.

For univariate cases, there are $(n+1)$ $M$ sets, namely $M(0),\ldots,M(n)$. We use a vector $(V(0),\ldots,V(n))$ to represent the numbers of unassigned minterms for the $(n+1)$ $M$ sets, where $V(i)$ records the number for the set $M(i)$. We call such a vector *problem vector*. Initially, the problem vector is equal to $(G(0),\ldots,G(n))$, given by the problem specification. As cubes are added into the on-set, the entries in the problem vector will be reduced. Eventually, when all the minterms have been decided, the problem vector will become a zero vector.

We can also represent a cube by a vector $[C(0),\ldots,C(n)]$ of length $(n+1)$, where $C(i)$ $(0 \le i \le n)$ represents the number of minterms of the cube in the set $M(i)$. We call such a vector *cube vector*. Note that in order to distinguish it from the problem vector, we represent the cube vector by square brackets. For example, assume that $n=2$ and $m=1$. The cube $X_1$ contains four minterms $(a_1,a_2,b_1) = (1,0,0)$, $(1,0,1)$, $(1,1,0)$, and $(1,1,1)$, as shown in Fig. 4a. The minterms $(1,0,0)$ and $(1,0,1)$ are in the set $M(1)$, the minterms $(1,1,0)$ and $(1,1,1)$ are in the set $M(2)$, and there are no minterms of the cube $X_1$ in the set $M(0)$. Therefore, the vector for the cube $X_1$ is $[0,2,2]$. Note that although each cube has a unique cube vector, a cube vector may correspond to several different cubes. For example, the cube $X_2$, as shown in Fig. 4b, has the same cube vector as the cube $X_1$.

| $Y_1\backslash X_1X_2$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | | | 1 | 1 |
| 1 | | | 1 | 1 |

(a) Cube $X_1$

| $Y_1\backslash X_1X_2$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | | 1 | | 1 |
| 1 | | 1 | | 1 |

(b) Cube $X_2$

Fig. 4: Two different cubes of the same cube vector $[0,2,2]$.

Our approach splits the problem vector into a set of cube vectors. In order to do this, it is important to study the valid form of a cube vector. We have the following claim.

**Theorem 1.** *Suppose a cube $C$ is composed of $u$ uncomplemented $X$-variables, $c$ complemented $X$-variables, and $l$ $Y$-variables, where $0 \le u,c \le n$, $u+c \le n$, and $0 \le l \le m$, then the cube vector is of the form*

$$2^{m-l} \times \left[0,\ldots,0,\binom{r}{0},\binom{r}{1},\ldots,\binom{r}{r},0,\ldots,0\right],$$

*where $0 \le r = n-u-c \le n$ and the cube vector has $u$ zeros at the beginning and $c$ zeros at the end.* $\square$

*Proof.* Consider the matrix representation of the cube. Since there are $l$ $Y$-variables in the cube, the number of missing $Y$-variables is $m-l$. Thus, the cube covers $2^{m-l}$ rows and all these rows cover the same set of columns. Therefore, we only need to show that for each row, it contains 0 items in the set $M(i)$ for any $0 \leq i < u$ and $n-c < i \leq n$, and $\binom{r}{i-u}$ items in the set $M(i)$ for any $u \leq i \leq n-c$.

Now, we consider the sub-cube $C_X$ which is composed of all the $X$-variables of the cube $C$. Let $S_n$ be the set of vectors $(a_1, \ldots, a_n) \in \{0,1\}^n$ that are contained in the sub-cube $C_X$. Since the number of uncomplemented $X$-variables in $C_X$ is $u$ and the number of missing $X$-variables is $r = n-u-c$, we have that for any $i = 0, \ldots, r$, there are $\binom{r}{i}$ vectors $(a_1, \ldots, a_n) \in S_n$ satisfying that $\sum_{i=1}^{n} a_i = u + i$. For any $0 \leq i < u$ or $n-c < i \leq n$, there are no vectors $(a_1, \ldots, a_n) \in S_n$ satisfying that $\sum_{i=1}^{n} a_i = i$. Therefore, the above claim on each row in the matrix representation of the cube $C$ is proved. □

**Example 2.** *Assume that $n = 3$ and $m = 2$. The cube $X_1Y_1$ contains 8 minterms $(a_1, a_2, a_3, b_1, b_2) = (1,0,0,1,0)$, $(1,0,0,1,1)$, $(1,0,1,1,0)$, $(1,0,1,1,1)$, $(1,1,0,1,0)$, $(1,1,0,1,1)$, $(1,1,1,1,0)$, and $(1,1,1,1,1)$. By definition, its cube vector is $[0,2,4,2]$. For this cube, the number of uncomplemented $X$-variables is 1, the number of complemented $X$-variables is 0, and the number of $Y$-variables is $l = 1$. By Theorem 1, the cube vector is $2^{2-1} \times [0, \binom{2}{0}, \binom{2}{1}, \binom{2}{2}]$, which is the same as that obtained by definition.* □

### B. The Basic Idea

The idea of our proposed method is to add cubes into the on-set of the Boolean function iteratively. Each time a cube is added, the number of unassigned minterms in the related $M$ sets will be reduced. Hence, the corresponding entries in the problem vector will be reduced. Eventually, the problem vector becomes a zero vector, and the Boolean function is constructed.

In general situations, adding an arbitrary cube to the on-set may cause intersection with the existing cubes in the on-set. However, in our method, we restrict that the cubes added in different iterations should be *disjoint* to each other. In this way, a cube added later does not contain any minterms belonging to cubes added before, and we can simply subtract the cube vector from the problem vector to update it after assigning a cube. We call this restriction *disjointness constraint*. Besides the benefit of updating the problem vector easily, this restriction also eliminates many redundant cases. For example, adding two non-disjoint cubes $X_1$ and $X_2$ is equivalent to adding two disjoint cubes $X_1$ and $\overline{X_1}X_2$. With disjointness constraint, only the latter situation is valid. Note that although the Boolean function is constructed by adding disjoint cubes, the final Boolean function will be further simplified by the two-level logic optimization tool ESPRESSO [31]. Thus, the final result is a set of non-disjoint cubes corresponding to a minimum SOP expression.

Our method picks one cube at each iteration. Since we will subtract the cube vector from the problem vector, we require that each entry in the cube vector for the cube should be no larger than the corresponding entry in the current problem vector. This constraint is called *capacity constraint*. If a cube satisfies both the disjointness constraint and the capacity constraint, we say the cube is *valid*.

In each iteration, we apply a greedy strategy to choose the cube to be added. We choose the largest cube among all valid cubes. One reason for this is that larger cubes have fewer literals. Since our primary goal is to minimize the literal count, we want to pick those largest cubes at the beginning, instead of expanding smaller cubes to larger ones by later added cubes. Another reason is that if we pick the largest cube in each iteration, the problem vector will be reduced to zero vector much faster, reducing the number of iterations.

It takes two steps to determine a cube to be added. First, we select the possible cube vectors for the largest cubes. The details will be discussed in Section IV-C. Second, we obtain the cubes for the candidate cube vectors. The details will be discussed in Section IV-D. Since there may exist multiple largest valid cubes at each iteration, therefore, even though we only consider the largest cubes, the potential solution space is still very large. We apply a heuristic breadth-first search method to find a good solution, which will be discussed in Section IV-E. Finally, given the runtime concern, we discuss a few speed-up techniques in Section IV-F.

It should be noted that the feasible solution space of the optimization problem is extremely large. Thus, it is computationally intractable to obtain an exact optimal solution. Nevertheless, the proposed method strives to explore a promising subset of the entire solution space to derive a good solution within a reasonable amount of time.

### C. Determining the Cube Vector for the Largest Valid Cubes

Suppose that at the beginning of one iteration, the problem vector is $(V(0), \ldots, V(n))$. Let $s$ be the sum of all the entries in the problem vector, i.e., $s = \sum_{i=0}^{n} V(i)$. Let $q = \lfloor \log_2 s \rfloor$. Since the largest valid cubes satisfy the capacity constraint, they contain at most $2^q$ minterms. Our method to find the largest valid cubes begins by checking whether there exist valid cubes with $2^q$ minterms.

Based on Theorem 1, the cube vector should be of the form $2^{m-l} \times [0,, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0]$, where $0 \leq r \leq n$, $0 \leq l \leq m$, and the vector has $0 \leq u \leq n-r$ zeros at the beginning. Moreover, we require that $m-l+r = q$, because the cube consists of $2^q$ minterms. Our method will examine all cube vectors that satisfy the above constraints on $r$, $l$, and $u$, and keep those which also satisfy the capacity constraint. For the kept cube vectors, we will find the corresponding cube assignments that satisfy the disjointness constraint. The details of how to check the existence of such a cube will be illustrated in Section IV-D. Such a cube is a largest valid cube. The following is an example of selecting a largest valid cube.

**Example 3.** *Suppose in a univariate case, $n = 2$, $m = 2$, and the initial problem vector is $(2,5,2)$. The sum of all the entries in the vector is 9. Thus, the maximum size of the valid cube is 8. We first check whether there exists any valid cube with 8 minterms. The cube vector for the cube should be $2^{m-l} \times [0,, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0]$, where $0 \leq r \leq 2$, $0 \leq l \leq 2$, $2-l+r = 3$, and the vector has $0 \leq u \leq 2-r$ zeros at the beginning. Given the requirements, we have $(l,r,u) = (0,1,0)$, $(0,1,1)$, or $(1,2,0)$. As a result, the possible cube vectors are $[4,4,0]$, $[0,4,4]$, and $[2,4,2]$. Although we have three cube vectors of size 8, only the cube vector $[2,4,2]$ satisfies the capacity constraint. Therefore, we will keep this vector and further find the corresponding cube assignments satisfying the disjointness constraint. Since we assume that $(2,5,2)$ is the initial problem vector, there are no cubes assigned yet. Thus, any cube assignment with the cube vector as $[2,4,2]$ satisfies the disjointness constraint. For example, we can choose the cube as $Y_1$. It is one of the largest valid cubes.* □

However, in some situations, it is impossible to find a valid cube with $2^q$ minterms, because all cubes of size $2^q$ violate either the capacity constraint or the disjointness constraint. The following shows an example.

**Example 4.** *Suppose in a univariate case, $n = 2$, $m = 3$, and the initial problem vector is $(1,6,2)$. The sum of all the entries in the vector is 9. Theoretically, the largest cube could have 8 minterms. The possible cube vectors of 8 minterms are $[0,0,8]$, $[0,8,0]$, $[8,0,0]$, $[0,4,4]$, $[4,4,0]$, and $[2,4,2]$.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2789725, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



*However, none of these vectors satisfy the capacity constraint. Thus, there is no valid cube of size* 8. □

If there exists no valid cube with $2^q$ minterms, our method will continue to check for smaller cubes. It will reduce the minterm number by half, and check whether there exists a valid cube with $2^{q-1}$ minterms. This procedure will be repeated until a valid cube with $2^i$ minterms for some $0 \leq i \leq q$ is found. This cube is a largest valid cube. In the worst case, the size of the cube is reduced to one. Since we can always find a valid cube of size 1, the procedure is guaranteed to terminate at some point.

We should note that in general cases, there could be more than one valid cube with the largest size. One reason is that there are more than one cube vector that satisfies the capacity constraint. Another reason is that for a specific cube vector, there could be multiple ways to assign the minterms. The following is an example.

**Example 5.** *Suppose in a univariate case,* $n = 2$, $m = 3$, *and the initial problem vector is* $(4, 8, 3)$. *The largest possible cube has* 8 *minterms. The cube vectors of size* 8 *that satisfy the capacity constraint are* $[0, 8, 0]$, $[4, 4, 0]$, *and* $[2, 4, 2]$. *Moreover, since* $(4, 8, 3)$ *is the initial problem vector, all the cubes with these cube vectors satisfy the disjointness constraint. For example, both the cubes* $X_1\overline{X_2}$ *and* $\overline{X_1}X_2$ *with the cube vector as* $[0, 8, 0]$ *are valid.* □

When there are more than one choice for the largest valid cube, we want to evaluate them and choose the best one. In Section IV-E, we will discuss an algorithm to traverse these choices and obtain a good choice.

### D. Obtaining Cubes for a Cube Vector

To obtain a largest valid cube, we further need to construct a cube for a candidate cube vector. We require the cube to satisfy the disjointness constraint. Since a cube is composed of $X$-variables and $Y$-variables, our procedure is divided into two steps: determining the sub-cube composed of $X$-variables and determining the sub-cube composed of $Y$-variables. For simplicity, we call them *$X$-cube* and *$Y$-cube*, respectively.

The $X$-cube is determined from the pattern of the cube vector. As shown in Theorem 1, if the vector is of the form $2^{m-l} \times [0, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0]$, where there are $u$ zeros at the beginning and $c$ zeros at the end, then the $X$-cube is composed of $u$ uncomplemented $X$-variables and $c$ complemented $X$-variables.

**Example 6.** *Suppose in a univariate case,* $n = 3$ *and we want to assign a cube for a potential cube vector* $[0, 2, 2, 0]$. *Since it has one zero at the beginning and one zero at the end, by Theorem 1, its $X$-cube should contain one uncomplemented $X$-variable and one complemented $X$-variable. Thus, the $X$-cube could be one of* $\overline{X_1}X_2$, $\overline{X_1}X_3$, $X_1\overline{X_2}$, $\overline{X_2}X_3$, $X_1\overline{X_3}$, *and* $X_2\overline{X_3}$. □

Next, for each candidate $X$-cube, we determine its associated $Y$-cube so that the cube obtained by ANDing the $X$-cube and $Y$-cube is disjoint to any of the cubes already added into the on-set. In the matrix representation, an $X$-cube specifies a set of columns and a $Y$-cube specifies a set of rows. The number of rows associated with the $Y$-cube is $2^{m-l}$. In order for the combination of the $X$-cube and the $Y$-cube to satisfy the disjointness requirement, the intersections of the rows associated with the $Y$-cube and the columns associated with the $X$-cube should all be 0. In other words, no minterms have been assigned to these intersections. Therefore, the $Y$-cube can only cover those rows that contain no ones at the intersections with the columns specified by the given $X$-cube.

**Example 7.** *Consider the case shown in Example 6, in which we want to assign a cube for the cube vector* $[0, 2, 2, 0]$. *We further assume that* $m = 3$ *and that* 12 *minterms have already been assigned as shown in Fig. 5. Suppose we are considering the $X$-cube* $\overline{X_1}X_3$, *which covers columns* 001 *and* 011. *Given the minterm assignment shown in the matrix, the possible rows that the $Y$-cube can cover are rows* 100, 101, 110, *and* 111. □

| $Y \backslash X$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 000 | | 1 | | 1 | | 1 | | 1 |
| 001 | | 1 | | 1 | | 1 | | 1 |
| 010 | | 1 | | | | 1 | | |
| 011 | | 1 | | | | 1 | | |
| 100 | | | | | | | | |
| 101 | | | | | | | | |
| 110 | | | | | | | | |
| 111 | | | | | | | | |

Fig. 5: Matrix representation for a univariate problem with $n = 3$ and $m = 3$.

The set of candidate rows that the $Y$-cube can cover can be represented as a Boolean function $F$, which includes all the minterms representing the candidate rows. For the situation shown in Example 7, the candidate rows form the Boolean function $F = Y_1$. In order to obtain a valid $Y$-cube, we only need to find a cube of size $2^{m-l}$ that is covered by the Boolean function $F$.

In our implementation, we use ESPRESSO [31] to get such a cube. We apply ESPRESSO to simplify the SOP $F$ constructed from the candidate rows for the $Y$-cube. If the size of the largest cube in the simplified SOP is smaller than $2^{m-l}$, then it means the largest cube covered by the Boolean function $F$ has a size smaller than $2^{m-l}$ and hence, we cannot assign a $Y$-cube for the given $X$-cube. Otherwise, there exists a valid $Y$-cube. Indeed, in most cases, the valid $Y$-cube is not unique. If we consider all the valid $Y$-cubes, we will obtain many largest valid cubes. To reduce the number of choices, in our implementation, we only pick one cube of size $2^{m-l}$ contained within the largest cubes returned by ESPRESSO. The cube we select is composed of the first $2^{m-l}$ minterms of the largest cube.

### E. Heuristic Breadth-First Search

As we discussed before, our method will choose the largest cube at each iteration. In many cases, there may exist multiple choices. However, among these choices, we cannot immediately know which one will lead to an expression with the least literals. Thus we will keep all of the choices and expand from them. This process will lead to a solution tree. An example of a solution tree for a univariate polynomial is shown in Fig. 6. The initial problem vector is $(4, 8, 2)$. For simplicity, we use a cube vector to represent a cube and we only distinguish branches by the different cube vectors that can be derived from each node. Each leaf of the solution tree corresponds to a final solution, represented by a set of cubes. Each internal node stores the remaining problem vector and a partial solution composed of a set of assigned cubes. The root node contains only the initial problem vector. At each internal node, the multiple choices of the largest valid cubes lead to multiple branches from the node.

In order to traverse the solution tree to obtain a good solution, we apply a breadth-first search. However, due to the exponential increase of the number of nodes with the level, our method does not expand all the nodes at each level. In contrast, it only expands a small set of local minimal nodes at each level.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2789725, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



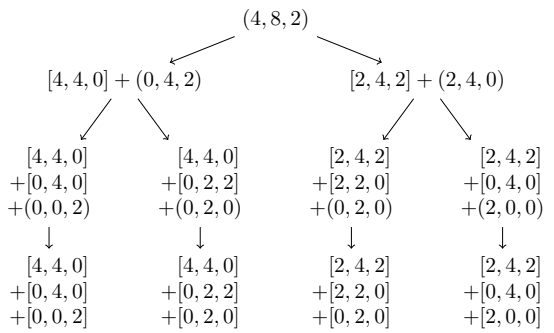


Fig. 6: An illustration of the solution tree for a univariate problem with the initial problem vector $(4, 8, 2)$ and $m = 2$.

As we just mentioned, each node in the solution tree may produce multiple branches, each corresponding to a largest valid cube that can be extracted from the remaining problem vector of the node, given the partial solution stored at that node. However, for different nodes at the same level, their largest valid cubes could have different sizes. In our proposed algorithm, for the next level, we only keep those nodes such that their newly added cubes are the largest among all the newly added cubes. For example, suppose that at the current level, we have three nodes $N_1$, $N_2$, and $N_3$. $N_1$ will branch to two nodes $M_1$ and $M_2$, each with a newly added cube of size 8. $N_2$ will branch to three nodes $M_3$, $M_4$, and $M_5$, each with a newly added cube of size 4. $N_3$ will branch to two nodes $M_6$ and $M_7$, each with a newly added cube of size 8. Then, at the next level, we will keep the nodes $M_1$, $M_2$, $M_6$, and $M_7$.

We only keep this set of nodes at the next level because for these nodes, the unassigned minterms are the fewest. For the previous example, suppose that the numbers of remaining minterms for the nodes $N_1$, $N_2$, and $N_3$ are 18. Then, for the nodes $M_1$, $M_2$, $M_6$, and $M_7$, their numbers of remaining minterms are all 10, while for the nodes $M_3$, $M_4$, and $M_5$, their numbers of remaining minterms are all 14. Moreover, by our procedure, the cubes added later are no larger than the ones added before. Since the largest cubes that can be extracted from the node $N_2$ are of size 4, this means the largest cubes that can be further extracted from $M_3$, $M_4$, or $M_5$ have a size at most 4. However, the largest cubes that can be further extracted from the nodes $M_1$, $M_2$, $M_6$, and $M_7$ could have a size of 8. Given this, the numbers of cubes in the final solutions from the nodes $M_1$, $M_2$, $M_6$, or $M_7$ are likely to be fewer than those in the final solutions from the nodes $M_3$, $M_4$, or $M_5$, and hence, the final solutions from the former are likely to have fewer literals than those from the latter.

Suppose the set of nodes at the next level with the largest added cubes is $S$. To further remove the unpromising candidates, we do a second round of filtering on the nodes in the set $S$. The criterion we use is the literal count of the partial solution formed by the set of assigned cubes at the node. For example, for the node $[4, 4, 0] + [0, 4, 0] + (0, 0, 2)$ in Fig. 6, its set of assigned cubes includes two cubes with cube vectors as $[4, 4, 0]$ and $[0, 4, 0]$, respectively. Then, we use the literal count of the partial solution formed by these two cubes as the filtering criterion. The literal count is obtained by ESPRESSO.

The second round of filtering works as follows. Assume that among all the nodes in the set $S$, the fewest literal count of the partial solution is $min$. Then, we only keep the nodes whose partial solutions have the literal counts less than or equal to $min + w$, where $w$ is a non-negative integer. Here, we apply a heuristic that a better partial solution is likely to lead to a better final solution. Therefore, we only keep nodes such that the literal counts for their partial solutions are close to the minimal value. Note that we do not just keep the nodes with the optimal partial solutions, i.e., the nodes such that the literal counts for the partial solutions are $min$, because it is possible that a better final solution is reached from a node with a suboptimal partial solution. By changing the value of $w$, we can trade off solution quality with runtime.

Algorithm 1 shows the proposed heuristic breadth-first search algorithm. It takes as inputs a problem vector $v$, a precision parameter $m$, and an optimization objective $obj$. The optimization objective can be anything that the user wants to optimize, not just restricted to the literal count of an SOP. For example, the objective can be set as the area, delay, or area-delay product of a multi-level circuit. Then, the algorithm will return a multi-level circuit which minimizes the specified objective.

The vector $Vec$ in the algorithm stores all the candidate nodes at one level that are to be expanded next. Each node $N$ has two entries: $N.cubeVec$, recording the remaining problem vector at the node $N$, and $N.cubeSet$, recording the set of assigned cubes at the node $N$.

The initial vector only contains a single node $N$ with $N.cubeVec$ as the input problem vector and $N.cubeSet$ empty (Lines 1–2). While the remaining cube vector of the first node in $Vec$ is not zero, the algorithm will obtain all the candidate nodes for the next level (Lines 3–14). For each candidate node $N$ at the current level, the algorithm calls the $findCubes$ function to extract a set $L$ of largest cubes from the node $N$ (Line 6). The function $findCubes$ realizes the idea discussed in Sections IV-C and IV-D; we will discuss the details of $findCubes$ shortly. Then, for each cube $C$ in the set $L$, a new node $N_{new}$ is created by including the cube $C$ into the current partial solution at the node $N$ (Lines 8–9). The new node $N_{new}$ is added into a vector $Vec_{new}$, which stores all the potential nodes for the next level (Line 10). Finally, the function $filter$ is called to obtain the set of candidate nodes for the next level that are to be expanded; the obtained set is assigned to $Vec$ again (Line 13). The function $filter$ implements the selection criteria we mentioned above: first, it only keeps the nodes such that their newly added cubes are the largest among all the newly added cubes, and then, it keeps the nodes such that the literal counts for their partial solutions are in the interval $[min, min + w]$.

Finally, when the loop terminates, the function $getBest$ is called to choose the best Boolean function from all the nodes in $Vec$ based on the specified optimization objective $obj$ (Line 15).

**Algorithm 1** The heuristic breadth-first search algorithm to find a good Boolean function.

**Input:** problem vector $v = (G(0), \ldots, G(n))$, an integer $m$, and an optimization objective $obj$.
**Output:** the final Boolean function $B$.

```
1: initialize a node N: N.cubeVec ⇐ v; N.cubeSet ⇐ ∅;
2: add the node N into an empty vector Vec;
3: while Vec[0].cubeVec ≠ 0 do
4:     Vector Vec_new ⇐ ∅;
5:     for each node N in Vec do
6:         cube set L ⇐ findCubes(N, m);
7:         for each cube C in L do
8:             N_new.cubeVec ⇐ N.cubeVec − cubeVec(C);
9:             N_new.cubeSet ⇐ N.cubeSet ∪ C;
10:            add the node N_new into Vec_new;
11:        end for
12:    end for
13:    Vec ⇐ filter(Vec_new);
14: end while
15: return getBest(Vec, obj);
```

One important function in our proposed algorithm is $findCubes$. It implements the proposed techniques discussed in Sections IV-C and IV-D to extract a set of largest valid cubes from a given node $N$. Algorithm 2 shows the details of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2789725, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



this function. Specifically, it initializes an empty set $L$ at the beginning (Line 1). This set will eventually store the largest valid cubes. Then, the procedure obtains a value $q$ so that $2^q$ is the maximal possible size of a valid cube (Line 2). Then, it enters into a loop to decide a set of largest valid cubes (Lines 3–15). In each iteration, it first decides a set $S$ of cube vectors of $2^q$ minterms that satisfy the capacity constraint specified by $N.cubeVec$ (Line 4). If the set $S$ is not empty, then for each vector in the set $S$, it calls the function $cubesFromVector$ to derive a set of cubes of cube vector $V$ that are disjoint to the partial assignment $N.cubeSet$, and adds that set to the set $L$ (Lines 5–9). The detailed flow of the function $cubesFromVector$ is shown in Algorithm 3, which implements the procedure described in Section IV-D. After all the vectors in the set $S$ are visited, if the set $L$ is not empty, then its stores a set of largest valid cubes and we simply return the set $L$ (Line 11). Otherwise, the value $q$ is decremented by 1 (Line 13), to further search for valid cubes of size $2^{q-1}$ in the next iteration.

**Algorithm 2** The procedure $findCubes$.

**Input:** a node $N$ and an integer $m$.
**Output:** a set of largest valid cubes that can be extracted from the node $N$.
1: $L \Leftarrow \phi$;
2: $q \Leftarrow \lfloor \log_2 sum(N.cubeVec) \rfloor$;
3: **while** true **do**
4: obtain the set $S$ of cube vectors of $2^q$ minterms that satisfy the capacity constraint specified by $N.cubeVec$;
5: **if** $S$ is not empty **then**
6: **for** each cube vector $V$ in $S$ **do**
7: $L \Leftarrow L \cup cubesFromVector(N, m, V)$;
8: **end for**
9: **end if**
10: **if** $L$ is not empty **then**
11: **return** $L$;
12: **else**
13: $q \Leftarrow q - 1$;
14: **end if**
15: **end while**

**Algorithm 3** The procedure $cubesFromVector$.

**Input:** a node $N$, an integer $m$, and a cube vector $V$.
**Output:** a set of cubes of cube vector $V$ that are disjoint to the partial assignment $N.cubeSet$.
1: $L \Leftarrow \phi$;
2: let $u$ and $c$ be the numbers of zeros at the beginning and the end of the vector $V$, respectively;
3: let $2^{m-l}$ be the multiplying factor of the vector $V$;
4: **for** each $X$-cube $C_X$ with $u$ uncomplemented and $c$ complemented $X$-variables **do**
5: **if** there exists a $Y$-cube $C_Y$ of size $2^{m-l}$ such that the cube $C_X \cdot C_Y$ is disjoint to the partial assignment $N.cubeSet$ **then**
6: find one such $Y$-cube $C_Y$;
7: add the cube $C_X \cdot C_Y$ into the set $L$;
8: **end if**
9: **end for**
10: **return** $L$;

### F. Speed-up Techniques

Although the heuristic breadth-first search algorithm removes some unpromising nodes in the solution tree from further expansion, there are still too many nodes to process as the degree of the polynomial increases, which increases the runtime considerably. In this section, we present two speed-up techniques.

*1) Removing Nodes with Duplicated Cube Sets:* For a node in the solution tree, even though the sum of all entries in its problem vector is in the interval $[2^q, 2^{q+1} - 1]$, the size of the largest valid cubes may not be $2^q$. Example 4 shows such a case. If this happens, we may add in sequence multiple cubes of the same size of $2^u$, where $u < q$ is an integer. In the original algorithm, different orders in which these cubes are added will produce different branches in the solution tree, but in most cases, these different branches will eventually lead to the same final result.

**Example 8.** *Consider the case shown in Example 4. Although the sum of all entries in the problem vector is* $9 \in [8, 15]$*, we cannot extract a valid cube of size* 8 *from the initial problem vector. As a result, the largest valid cubes contain* 4 *minterms. The associated cube vector is either* $[1, 2, 1]$ *or* $[0, 4, 0]$*. In the original algorithm, if the first assigned cube is obtained from the cube vector* $[1, 2, 1]$*, then the second one will be obtained from* $[0, 4, 0]$*. On the other hand, if the first assigned cube is obtained from* $[0, 4, 0]$*, then the second one will be obtained from* $[1, 2, 1]$*. Thus, there will be two branches in the solution tree. However, these two branches will produce the same results.* □

These different branches caused by different orders in selecting the cubes are unnecessary to be explored again. To remove them, we keep track of the sets of cube vectors that we have already processed. If the set of cube vectors at the current node has been examined before, the node will be discarded.

*2) Limiting the Number of $X$-cubes:* As we mentioned in Section IV-D, to construct a cube for a potential cube vector, we will first enumerate all the $X$-cubes for that cube vector. However, when we extract the largest cubes from the root node in the solution tree, we do not need to enumerate all the $X$-cubes for a valid cube vector. We only need to choose one $X$-cube for the cube vector. The reason is that at this moment, we have not selected any cubes yet. Since two $X$-cubes for the same cube vector have the same numbers of complemented and uncomplemented $X$-variables, they are equivalent under the permutation of the $X$-variables.

However, when we extract the largest cubes from the later nodes in the solution tree, we have to enumerate all the $X$-cubes for a given cube vector. Although these $X$-cubes by themselves are equivalent under the permutation of the $X$-variables, when they are combined with the already selected cubes, they form different Boolean functions even if variable permutation is considered. Nevertheless, in our implementation, in order to reduce the runtime, we do not enumerate all the $X$-cubes. Instead, we keep $h$ of them. The larger the value of $h$ we choose, the more nodes in the solution tree are explored and hence, the better the final result is. However, this increases runtime. Therefore, by changing the value of $h$, we can trade off solution quality with runtime.

## V. Synthesis of Multivariate Polynomials

In Section IV, we describe the proposed method for univariate polynomials. In this section, we present extensions to handle arbitrary multivariate polynomials.

The procedure to synthesize Boolean function for multivariate polynomials is the same as that for univariate polynomials. However, the form of a cube vector and the way to determine the valid cube vectors for a problem vector are different. In the remaining of this section, we describe these major differences and some associated changes.

For multivariate cases, the problem vector is defined as $(V(0, \ldots, 0), \ldots, V(d_1, \ldots, d_k))$, where for any $0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$, $V(s_1, \ldots, s_k)$ represents the number of unassigned minterms in the set $M(s_1, \ldots, s_k)$. The cube vector is defined as $[C(0, \ldots, 0), \ldots, C(d_1, \ldots, d_k)]$, where for any $0 \leq s_1 \leq d_1, \ldots, 0 \leq s_k \leq d_k$, $C(s_1, \ldots, s_k)$ represents the number of minterms of the cube that are in the set $M(s_1, \ldots, s_k)$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2789725, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



First, we have the following theorem on the form of a cube vector.

**Theorem 2.** *Suppose a cube $C$ is a product of $(k+1)$ sub-cubes $C_{X_1},\ldots,C_{X_k},C_Y$. For $1\le i\le k$, the sub-cube $C_{X_i}$ is composed of $u_i$ uncomplemented $X_{i,\cdot}$-variables and $c_i$ complemented $X_{i,\cdot}$-variables, where $0\le u_i,c_i\le d_i$ and $u_i+c_i\le d_i$. The sub-cube $C_Y$ is composed of $l$ $Y$-variables, where $0\le l\le m$. For each $1\le i\le k$, define a vector $[C_i(0),C_i(1),\ldots,C_i(d_i)]$ such that for any $0\le j<u_i$ or $d_i-c_i<j\le d_i$, $C_i(j)=0$ and for any $u_i\le j\le d_i-c_i$, $C_i(j)=\binom{d_i-u_i-c_i}{j-u_i}$. Then, for any $0\le s_1\le d_1,\ldots,0\le s_k\le d_k$, $C(s_1,\ldots,s_k)=2^{m-l}\prod_{i=1}^{k}C_i(s_i)$.* □

*Proof.* First, for any $1\le i\le k$ and $0\le s_i\le d_i$, consider the number of vectors $(a_{i,1},\ldots,a_{i,d_i})\in\{0,1\}^{d_i}$ that are contained in the sub-cube $C_{X_i}$ and satisfy that $\sum_{j=1}^{d_i}a_{i,j}=s_i$. Denote that number as $F_i(s_i)$. Since the cube $C_{X_i}$ contains $u_i$ uncomplemented $X_{i,\cdot}$-variables and $c_i$ complemented $X_{i,\cdot}$-variables, for any $u_i\le s_i\le d_i-c_i$, we have $F_i(s_i)=\binom{d_i-u_i-c_i}{s_i-u_i}$, while for any $0\le s_i<u_i$ or $d_i-c_i<s_i\le d_i$, we have $F_i(s_i)=0$. In summary, for any $0\le s_i\le d_i$, we have $F_i(s_i)=C_i(s_i)$.

By the definition of $C(s_1,\ldots,s_k)$, it denotes the number of vectors $(a_{1,1},\ldots,a_{k,d_k},b_1,\ldots,b_m)\in\{0,1\}^{n+m}$ such that they are contained in the cube $C$ and for all $1\le i\le k$, $\sum_{j=1}^{d_i}a_{i,j}=s_i$. By the above analysis on the sub-cube $C_{X_i}$ and the assumption that the number of $Y$-variables is $l$, we have for any $0\le s_1\le d_1,\ldots,0\le s_k\le d_k$,

$$C(s_1,\ldots,s_k)=2^{m-l}\prod_{i=1}^{k}F_i(s_i)=2^{m-l}\prod_{i=1}^{k}C_i(s_i).$$

□

**Example 9.** *Assume in a multivariate case, we have $k=2$, $d_1=2$, $d_2=3$, and $m=2$. Consider the cube $\overline{X_{1,2}}X_{2,2}Y_1$. We first obtain its $C(s_1,s_2)$ values by the definition. The cube covers 16 minterms of the form $(a_{1,1},0,a_{2,1},1,a_{2,3},1,b_2)$, where $a_{1,1},a_{2,1},a_{2,3},b_2\in\{0,1\}$. The minterms of the cube that are in the set $M(0,1)$ is $(0,0,0,1,0,1,0)$ and $(0,0,0,1,0,1,1)$. Therefore, $C(0,1)=2$. Similarly, we can obtain $C(0,2)=4$, $C(0,3)=2$, $C(1,1)=2$, $C(1,2)=4$, $C(1,3)=2$, and $C(s_1,s_2)=0$, for all $(s_1,s_2)$ such that either $s_1=2$ or $s_2=0$.*

*Now, we obtain the $C(s_1,s_2)$ values by Theorem 2. For the sub-cube $C_{X_1}=\overline{X_{1,2}}$, we have $u_1=0$ and $c_1=1$. For the sub-cube $C_{X_2}=X_{2,2}$, we have $u_2=1$ and $c_2=0$. For the sub-cube $C_Y=Y_1$, we have $l=1$. The vector $[C_1(0),C_1(1),C_1(2)]=[1,1,0]$ and the vector $[C_2(0),C_2(1),C_2(2),C_2(3)]=[0,1,2,1]$. By Theorem 2, we have*

$$C(0,0)=2C_1(0)C_2(0)=0,\quad C(0,1)=2C_1(0)C_2(1)=2,$$
$$C(0,2)=2C_1(0)C_2(2)=4,\quad C(0,3)=2C_1(0)C_2(3)=2,$$

*which are the same as what we obtained by the definition. The other $C(s_1,s_2)$ values obtained by Theorem 2 are also the same as what we obtained by the definition.* □

The procedure to handle a multivariate case is similar to the univariate case. In each iteration, we determine the largest valid cubes that can be extracted from the current problem vector. In order to do this, we first determine the largest valid cube vectors and then map them into the largest valid cubes. To determine the cube vector, it reduces to determine the number of uncomplemented variables, $u_i$, and the number of complemented variables, $c_i$, in each sub-cube $C_{X_i}$ and the number of $Y$-variables, $l$, in the sub-cube $C_Y$. Suppose we are checking whether there exists a valid cube of size $2^q$. Then, we have

$$m-l+\sum_{i=1}^{k}(d_i-u_i-c_i)=q. \tag{5}$$

Furthermore, $u_1,c_1,\ldots,u_k,c_k$, and $l$ should satisfy

$$0\le u_i,c_i\le d_i,u_i+c_i\le d_i,\text{for all } i=1,\ldots,k, \tag{6}$$
$$0\le l\le m. \tag{7}$$

We will enumerate all sets of $u_1,c_1,\ldots,u_k,c_k,l$ that satisfy Eqs. (5), (6), and (7). For each set, we apply Theorem 2 to obtain the cube vector and check whether it satisfies the capacity constraint. If the capacity constraint is satisfied, then we will record the set of $u_1,c_1,\ldots,u_k,c_k,l$ as a valid set, which will be further mapped to the largest cubes. The following example shows how we determine the valid sets of parameters $u_1,c_1,\ldots,u_k,c_k,l$ for a multivariate problem.

**Example 10.** *Assume in a multivariate case, we have $k=2$, $d_1=2$, $d_2=1$, and $m=3$. The initial problem vector is $(G(0,0),G(0,1),G(1,0),G(1,1),G(2,0),G(2,1))=(2,3,9,10,12,10)$. The sum of all the entries is 46. Thus, the theoretic largest cubes cover $2^5=32$ minterms. We first check whether there exists any valid set of parameters corresponding to a cube of size 32. The requirements on the parameters $u_1,c_1,u_2,c_2,l$ are*

$$3-l+2-u_1-c_1+1-u_2-c_2=5,$$
$$0\le u_1,c_1,\le 2,u_1+c_1\le 2,$$
$$0\le u_2,c_2\le 1,u_2+c_2\le 1,$$
$$0\le l\le 3.$$

*Solving the above equations, we obtain five sets of parameters:*

$$(u_1,c_1,u_2,c_2,l)=(1,0,0,0,0),(0,1,0,0,0),(0,0,1,0,0),$$
$$(0,0,0,1,0),(0,0,0,0,1).$$

*The corresponding cube vectors are*

$$[C(0,0),C(0,1),C(1,0),C(1,1),C(2,0),C(2,1)]=$$
$$[0,0,8,8,8,8],[8,8,8,8,0,0],[0,8,0,16,0,8],$$
$$[8,0,16,0,8,0],[4,4,8,8,4,4].$$

*Among them, only the first one satisfies the capacity constraint. Therefore, there exists one valid set of parameters, which is $(u_1,c_1,u_2,c_2,l)=(1,0,0,0,0)$. The corresponding cubes only contain an uncomplemented $X_{1,\cdot}$-variable.* □

For each valid set of parameters $u_1,c_1,\ldots,u_k,c_k,l$, we further construct all possible cubes satisfying these parameters and the disjointness constraint. Same as the univariate case, we first determine the $X$-cube and then the $Y$-cube. For multivariate case, to determine the $X$-cube, we need to determine all sub-cubes $C_{X_1},\ldots,C_{X_k}$. Note that given $u_i$ and $c_i$, the corresponding sub-cube $C_{X_i}$ is not unique. All possible $X$-cubes should be built by all the combinations of the possible sub-cubes $C_{X_1},\ldots,C_{X_k}$. However, in our implementation, same as the univariate case, we limit the number of $X$-cubes we consider to $h$, due to runtime concern. Each chosen $X$-cube specifies a set of columns in the matrix representing the Boolean function. We apply the same method shown in Section IV-D to determine a valid $Y$-cube associated with each chosen $X$-cube.

## VI. Complexity Analysis

In this section, we analyze the time and space complexity of our method. The analysis is general for an arbitrary multivariate target polynomial.

Our method constructs a search tree of the form shown in Fig. 6. Its time complexity is proportional to the product of the amount of work at each node and the number of nodes in

the search tree. We first analyze the amount of work at each node. At each node, our algorithm extracts a set of valid largest cubes from the remaining problem vector of the node. This procedure essentially enumerates all the valid $X$-cubes. For each $X$-cube, it further decides a single valid $Y$-cube. Thus, the amount of work is equal to the number of enumerated valid $X$-cubes times the amount of work in deciding a single valid $Y$-cube. The number of valid $X$-cubes enumerated is no more than the number of all possible $X$-cubes. Given that there are $n$ $X$-variables in total, the total number of $X$-cubes is $3^n$, because each $X$-variable could appear in the complemented form, appear in the uncomplemented form, or not appear. For each valid $X$-cube, the deciding of an associated $Y$-cube involves manipulating the 1's stored in the $2^n$-by-$2^m$ matrix representing the partial solution at the node. The runtime of this manipulation dominates the runtime of deciding a $Y$-cube and is bound by a constant times the total number of entries in the matrix (i.e., $2^{n+m}$). Therefore, the amount of work at each node is bounded by $c_1 \cdot 3^n \cdot 2^{n+m}$, where $c_1$ is a constant.

Next, we analyze the number of nodes in the search tree. Due to our strategy of keeping the local minimal at each level of the search tree, the number of nodes at each level can be treated as a constant. Thus, the number of nodes in the search tree is proportional to the number of levels of the search tree. Now, we analyze the number of levels of the search tree. Suppose the sum of all the entries of the initial problem vector is $S$. Due to our strategy of always choosing the largest available cube, for a usual case, the number of levels of the search tree is equal to the number of ones in the binary representation of $S$. Note that since $S \leq 2^{n+m}$, the total number of levels is bounded by $(n+m)$. Thus, the number of nodes in the search tree is bounded by $c_2(n+m)$, where $c_2$ is a constant. In conclusion, the total amount of work of our procedure is bounded by $c_1c_2(n+m)3^n \cdot 2^{n+m}$. Therefore, the runtime is $O((n+m)3^n \cdot 2^{n+m})$.

In our implementation, the maximal space needed is to store all the candidate nodes at the next level, which are expanded from the nodes at the current level. By the above analysis, the number of nodes at the current level is a constant. For each node at the current level, the number of candidate nodes expanded from it is bounded by $3^n$. Thus, the total number of candidate nodes at the next level is $O(3^n)$. Note that in our implementation, for the ease of manipulation, each candidate node at the next level is associated with a $2^n$-by-$2^m$ matrix recording the partial assignment of 1's. This matrix of size $O(2^{n+m})$ dominates the storage of a candidate node. Therefore, the maximal space consumption is $O(3^n \cdot 2^{n+m})$.

Note that although our method is exponential in time and space, the runtime and memory consumption of our algorithm are still affordable for a normal stochastic circuit, since the values $n$ and $m$ for a typical stochastic circuit tend to be small.

## VII. Experimental Results

In this section, we show the experimental results of the proposed method. All the experiments were conducted on a desktop with 3.20 GHz Intel® Core™ i5-4570 CPU and 16.0 GB RAM. ESPRESSO [31] was used to evaluate the literal count and ABC [32] was used to synthesize the multi-level circuits.

In Section VII-A, we first study the effect of our proposed heuristic breadth-first search strategy. We compare it with a branch-and-bound strategy proposed in the preliminary version of this work [21]. In Sections VII-B and VII-C, we compare our method to previous state-of-the-art methods. Our method can synthesize general polynomials. In Section VII-B, we compare our method to a state-of-the-art method that can also synthesize general polynomials, i.e., the method proposed in [16]. In Section VII-C, we compare our method to a method that is efficient in synthesizing some commonly used arithmetic functions, i.e., the method proposed in [17].

### A. Effect of the Proposed Breadth-first Search

In this work, we proposed a heuristic breadth-first search algorithm to explore a small but promising subset of the solution space. In a preliminary version of this work [21], a branch-and-bound-based algorithm was proposed to explore the solution space. In this section, we demonstrate the advantage of the breadth-first search strategy over the previous branch-and-bound strategy. We compared the runtime and the literal count of the final solution of both methods. The test cases are univariate targets with degree $n = 3$ and precision parameter $m = 3, \ldots, 7$. For each pair of $n$ and $m$, 50 random problem vectors were generated as the inputs and the average results of these 50 cases were presented. Our algorithm has two parameters $w$, which is used to filter the unpromising nodes, and $h$, which limits the number of $X$-cubes to be considered for each valid cube vector. In this experiment, we chose $w = 2$ and $h = 5$. The comparison result is shown in Fig. 7, where the solid line shows the speed-up ratio of the breadth-first search over the branch-and-bound algorithm ($y$ axis on the left) and the dotted line shows the relative literal increase of the breadth-first search over the branch-and-bound algorithm ($y$ axis on the right).

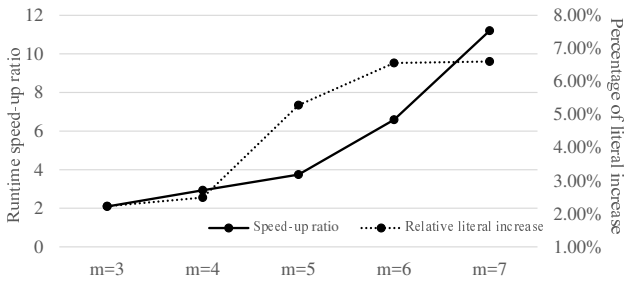


Fig. 7: The speed-up ratio and relative literal increase of the breadth-first search algorithm over the branch-and-bound algorithm [21].

We can see that the breadth-first search algorithm is much faster than the branch-and-bound algorithm. As the circuit size increases, the speed-up ratio also increases. Meanwhile, its quality loss, measured by the percentage of literal increase, is small. For all pairs of $n$ and $m$, the literal increase is less than 7%. Therefore, we can see that the proposed breadth-first search method dramatically improves the runtime with negligible quality loss compared to the previous branch-and-bound method.

### B. Synthesizing Random General Polynomials

In this section, we use random general polynomials as the design targets to study the effect of our proposed method. We compared it with the method in [16], a state-of-the-art method in synthesizing general polynomials. We show the results for synthesizing both the univariate and the multivariate polynomials.

*1) Univariate Cases:* In this section, we studied our proposed method for synthesizing univariate polynomials. We applied our method to different univariate problems with degree $n = 3, \ldots, 7$ and precision parameter $m = 3, \ldots, 7$. For each pair of $n$ and $m$, 50 random problem vectors were generated as the inputs and the average results of these 50 cases were presented. We chose the parameters $w = 2$ and $h = 5$.

Fig. 8 shows the literal count reduction by our method over the previous method [16]. A positive value indicates a reduction compared to the previous method, while a negative value indicates an increase. It can be seen that compared to the previous method, the proposed method reduces the literal count for all combinations of $n$ and $m$. When $n$ is small, the literal count reduction is small. It is because the previous greedy method is able to find a good solution among limited

choices. However, as $n$ increases, more percentage of literals is saved. For $n = 7$, the literal saving reaches up to 28%.

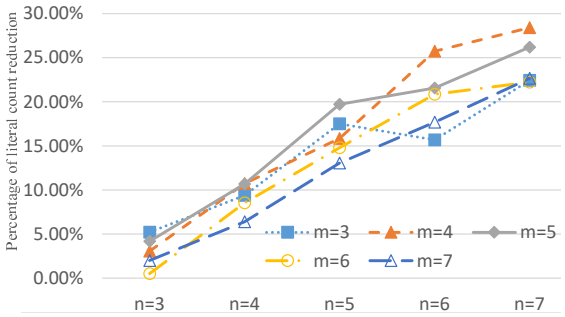


Fig. 8: The average percentage of literal count reduction by the proposed method over the previous method [16] for univariate problems.

Although the primary target of our proposed method is two-level circuits, it can be adapted to optimize multi-level circuits by setting the $obj$ parameter in Algorithm 1 as a measure for multi-level circuits. To study the effectiveness of our proposed method in synthesizing multi-level circuits, we applied ABC [32] within the $getBest$ function in Algorithm 1 to synthesize a multi-level design for each candidate Boolean function in the final node vector $Vec$. We chose the objective as the area-delay product, since for a multi-level circuit, there is a trade-off between its area and delay. Therefore, the $getBest$ function returns a circuit with the minimal area-delay product among all candidate Boolean functions in the final node vector $Vec$.

Figs. 9, 10, and 11 plot the average percentage of reduction in area, delay, and area-delay product, respectively, of the multi-level circuits produced by the proposed method over the previous method [16]. As we can see, for all the cases, our proposed method reduces the delay and the area-delay product, and for most cases, our method also reduces the area. Overall, as the degree $n$ increases, the reduction in area, delay, and area-delay product also increases. For $n = 7$, the reduction in area-delay product reaches up to 25%.

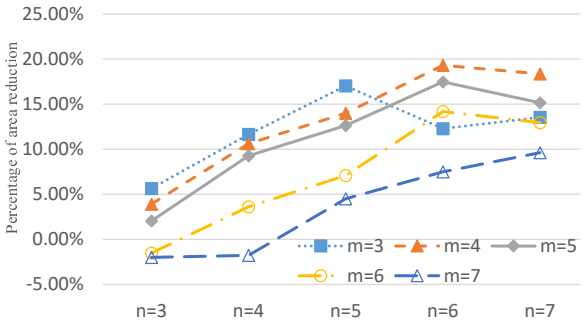


Fig. 9: The average percentage of area reduction of the multi-level circuits produced by the proposed method over the previous method [16] for univariate problems.

Finally, we studied the effects of different choices of the parameters $w$ and $h$. We chose $w = 0, 1, 2$ and $h = 1, 5$. We applied each parameter combination to the same set of test cases used before. For each parameter combination, we obtained the average area-delay product improvement, literal count improvement, and runtime over all the cases. The results for different combinations of $w$ and $h$ are shown in Fig. 12. From the figure, we can see that as $w$ increases, the circuit quality and the runtime does not change too much. This indicates that for univariate cases, the final optimal solution is very likely to be derived from one of the optimal partial solutions. However, when $h$ increases from 1 to 5,

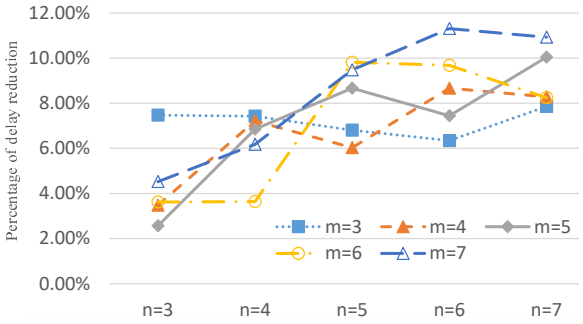


Fig. 10: The average percentage of delay reduction of the multi-level circuits produced by the proposed method over the previous method [16] for univariate problems.

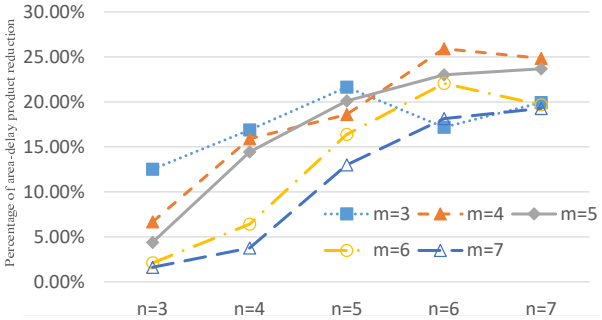


Fig. 11: The average percentage of area-delay product reduction of the multi-level circuits produced by the proposed method over the previous method [16] for univariate problems.

the circuit quality has a much larger increase at the cost of longer runtime. This is because more $X$-cubes are selected for each valid cube vector and hence, a larger solution space is explored. Thus, we can conclude that for univariate cases, an effective way to improve the circuit quality is to choose more number of $X$-cubes for each valid cube vector. The parameter combination $(w, h) = (2, 5)$ has the longest runtime among all the combinations. For this parameter choice, the average runtime is below 2 minutes.

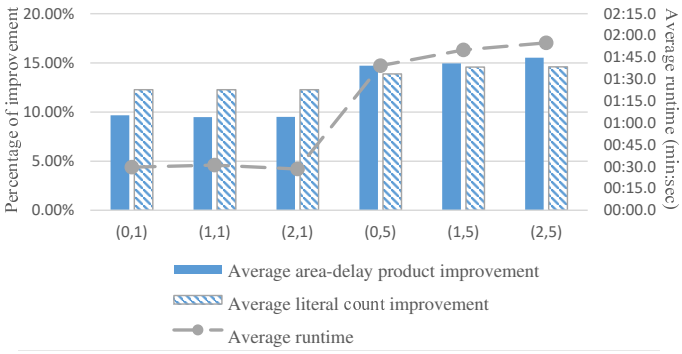


Fig. 12: The average area-delay product improvement, literal count improvement, and runtime with different parameter combinations $(w, h)$ for univariate problems.

*2) Multivariate Cases:* In this section, we studied our proposed method for synthesizing multivariate polynomials. We used bivariate polynomials (i.e., $k = 2$) as the test cases. We generated 100 groups of problem vectors for bivariate polynomials, with each group characterized by a tuple $(d_1, d_2, m)$, where $d_1$ and $d_2$ are the degrees of the variables $x_1$ and $x_2$ in the given polynomial, respectively. We chose

the degree sum $n = d_1 + d_2$ from 3 to 7 and the precision parameter $m$ from 3 to 7. For each $3 \leq n \leq 7$, we chose $d_1$ from 1 to $(n-1)$ and set $d_2 = n - d_1$. Thus, the total number of different $(d_1, d_2, m)$ tuples is $(2+3+4+5+6) \times 5 = 100$. For each tuple $(d_1, d_2, m)$, 50 random problem vectors were generated as the inputs and the average results of these 50 cases were presented. We chose the parameters $w = 2$ and $h = 5$.

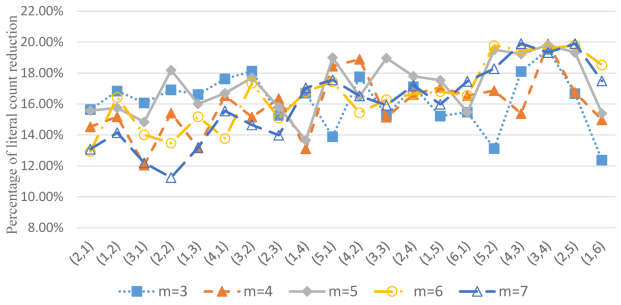


Fig. 13: The average percentage of literal count reduction by the proposed method over the previous method [16] for bivariate problems.

Fig. 13 shows the average percentage of literal count reduction by the proposed method over the previous method [16]. Different markers correspond to different parameters $m$, ranging from 3 to 7. The horizontal labels mark the degree pairs $(d_1, d_2)$. Fig. 13 shows that compared to the previous method, our method achieves literal count reduction for all tuples $(d_1, d_2, m)$. For some tuples, the literal count reduction reaches 20%. However, unlike univariate cases, there is no trend that as $n = d_1 + d_2$ increases, the percentage of literal count reduction increases.

We also studied the performance of our method in synthesizing multi-level circuits. The experiment setup was similar to that for the univariate cases. Due to the space limit, we only show the average percentage of reduction in area-delay product of the multi-level circuits produced by our proposed method over the previous method [16]. The result is shown in Fig. 14. We can see that for all the tuples, our method reduces the circuit area-delay product, compared to the previous method. For $n = 7$, the reduction in the area-delay product reaches up to 17%, while for $m = 7$, the reduction reaches up to 11%.

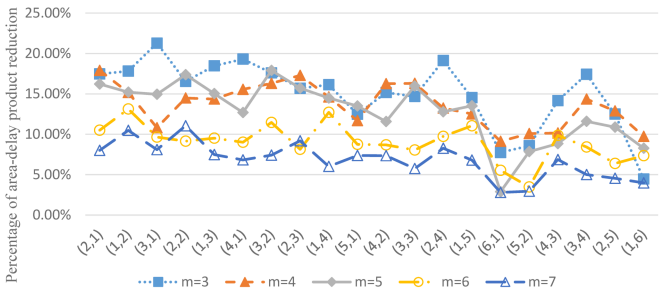


Fig. 14: The average percentage of area-delay product reduction of the multi-level circuits produced by the proposed method over the previous method [16] for bivariate problems.

We also studied the effects of different choices of the parameters $w$ and $h$. We chose four parameter combinations, which are $(w, h) = (0, 1)$, $(1, 1)$, $(2, 1)$, and $(2, 5)$. The other parts of the experiment setup are similar to those for the univariate cases. Fig. 15 shows the average area-delay product improvement, literal count improvement, and runtime for these four parameter combinations. We can see that as $w$ changes from 0 to 2, the circuit quality improves, which is different from the univariate cases. It is because for bivariate cases, the solution tree has much more branches and therefore, the optimal final solution is more likely to be derived from a suboptimal partial solution. As $w$ increases, more suboptimal partial solutions are explored, hence leading to a better final solution. When the parameter $h$ changes from 1 to 5, the circuit quality also improves at the cost of longer runtime, which is similar to what we observed for the univariate cases. From Fig. 15, it is also interesting to note that the improvement in the area-delay product is smaller than that in the literal count for each parameter combination. We believe the reason is because for bivariate cases, as the circuits become more complicated than those for univariate cases, the correlation between the quality of a two-level design and that of a multi-level design becomes weaker. This calls for a more powerful synthesis method for stochastic circuits that directly targets at multi-level designs, which we will study in our future work.

Finally, we compared the accuracy and the runtime of our method to those of the previous method [16]. For a combinational stochastic circuit, its output accuracy depends on the error between the target function and the stochastic function realized by the stochastic circuit and the length of the stochastic bit streams [2]. Given the same target function and the same precision parameter $m$, the functions realized by the stochastic circuits produced by our method and by the previous method [16] are the same. Therefore, with the same stochastic bit stream length, the two methods produce circuits with the same accuracy. In terms of runtime, the previous method [16] is much faster than ours, because it does not search the feasible solution space. Indeed, the previous method only takes time to fill 1's into a matrix of size $2^n$ by $2^m$ and hence, has runtime of $O(2^{n+m})$. By our analysis in Section VI, the runtime of our method is $O((n+m)3^n \cdot 2^{n+m})$. Thus, our method takes much more time than the previous method [16]. However, this is unavoidable in order to have a decent search of the solution space. Nevertheless, the runtime of our algorithm is still affordable: as shown in Figs. 12 and 15, the average runtime for the parameter combination $(w, h) = (2, 5)$ is on the scale of several minutes. In situations where better circuit quality is pursued, our method gives a better solution under a reasonable amount of runtime.

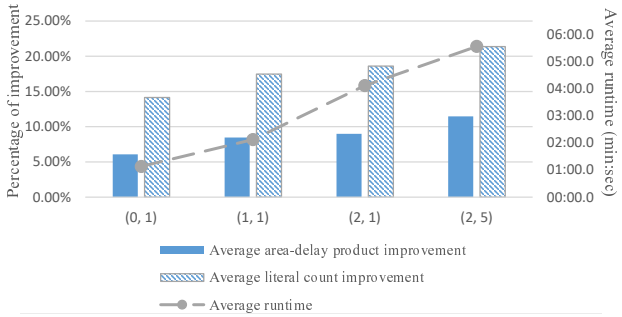


Fig. 15: The average area-delay product improvement, literal count improvement, and runtime with different parameter combinations $(w, h)$ for bivariate problems.

### C. Synthesizing Commonly Used Arithmetic Functions

In this section, we applied our method to synthesize some commonly used arithmetic functions, such as trigonometric, exponential, and logarithmic functions. We compared our method to the method in [17], a state-of-the-art method in synthesizing these commonly used functions.

The original target functions are shown in column 1 of Table I. Same as [17], we used the Maclaurin series expansions of these functions as our synthesis target polynomials. The degrees of these Maclaurin expansions are listed in column 2 of the table. Besides, same as [17], a scaling of $1/\pi$ is

applied to the Maclaurin expansion of the function $\sin(\pi x)$. These target polynomials are all univariate polynomials.

Before applying our approach, we performed a simple transformation on the input Maclaurin expansion when possible. We use $\sin(x)$ as an example. The 7-th order Maclaurin expansion of $\sin(x)$ is

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} = x \cdot g(x^2),$$

where $g(t) = 1 - t/3! + t^2/5! - t^3/7!$. Our proposed approach is applied to the function $g(t)$. After we obtain a stochastic implementation of $g(t)$, we replace each input for $t$ by an AND gate with two inputs as $x$, which implements $x^2$. Then, the output of the circuit is $g(x^2)$. Finally, we multiply the output by an $x$ using an extra AND gate. This eventually gives $xg(x^2)$, the 7-th order Maclaurin expansion of $\sin(x)$. This kind of transformation was also applied to $\cos(x)$, $\tanh(x)$, and $\sin(\pi x)$ in Table I.

We compared our designs to the designs synthesized by the approach in [17]. In [17], a factorization-based method is applied to the Maclaurin expansion of a target function. The designs in [17] include some constant input probabilities. However, as we mentioned in Section III-A, the generation of these constant input probabilities is not free. Typically, they are generated from source input probabilities of $0.5$ by additional conversion circuits. For a fair comparison, we used a method proposed in [15] to synthesize the optimal conversion circuits. Also, the designs in [17] include some delay elements to generate roughly independent $x$ input bit streams. For a fair comparison, we converted the designs in [17] into combinational circuits by removing the delay elements and adding the AND gates that generate the power terms of $x$ when proper.

For both methods, the precision $m$ was chosen as $8$. Once a circuit was constructed, ABC was further applied to optimize it to obtain the final area and delay. The area, delay, and area-delay product of the two methods are listed in columns 4, 5, and 6 of Table I, respectively. The rows with "Cube" and "Factor" correspond to our method and the method in [17], respectively. Column 7 lists the ratio of the area-delay product of our method over that of the previous method. We can see that the circuits obtained by our method have smaller area-delay products than the previous method for most of the functions. The geometric mean of the area-delay product ratio between the two methods is 0.806, which indicates that on average, our method can reduce the area-delay product by 20% over the method in [17].

TABLE I: Comparison between our method and the method in [17] in synthesizing some commonly used arithmetic functions.

| Function | Degree | | Area | Delay | Area-delay product | Product ratio | MAE |
|---|---|---|---|---|---|---|---|
| $\sin(x)$ | 7 | Cube | 43 | 6.7 | 288.1 | 0.686 | 0.0106 |
| | | Factor | 56 | 7.5 | 420.0 | | 0.0107 |
| $\cos(x)$ | 8 | Cube | 27 | 4.7 | 126.9 | 0.332 | 0.0075 |
| | | Factor | 58 | 6.6 | 382.8 | | 0.0079 |
| $\tanh(x)$ | 9 | Cube | 74 | 6.4 | 473.6 | 0.992 | 0.0115 |
| | | Factor | 62 | 7.7 | 477.4 | | 0.0113 |
| $\log(1+x)$ | 5 | Cube | 49 | 6.3 | 308.7 | 0.757 | 0.0172 |
| | | Factor | 48 | 8.5 | 408.0 | | 0.0173 |
| $e^{-x}$ | 5 | Cube | 67 | 6.6 | 422.2 | 1.082 | 0.0109 |
| | | Factor | 47 | 8.3 | 390.1 | | 0.0104 |
| $\sin(\pi x)$ | 9 | Cube | 76 | 6.6 | 501.6 | 1.484 | 0.0312 |
| | | Factor | 52 | 6.5 | 338.0 | | 0.0318 |

We also compared the accuracy of the two methods. We chose the length of the stochastic bit streams as $1024$. For each function $f(x)$, we chose 9 input points $x = 0.1, 0.2, \ldots, 0.9$ for simulation. For each input point, the stochastic circuit of a target function was simulated for $100$ times. The mean absolute error (MAE) was obtained over all input points and simulation runs. The MAE's for the two methods are listed in column 8 of Table I. The result shows that the two methods have almost the same accuracy. This is because they implement the same Maclaurin expansion for each arithmetic function. The differences are due to the randomness in the simulation.

## VIII. Conclusion

In this work, we proposed a method based on cube assignment to synthesize general stochastic circuits. Different from the traditional logic synthesis problems, for stochastic circuit synthesis, there exist many different Boolean functions that perform the same computation. Primarily targeting at two-level designs, we proposed a heuristic breadth-first search algorithm to explore the solution space. The experimental results showed that our algorithm can synthesize SOPs with much fewer literals than a previous state-of-the-art method. A straightforward mapping of the obtained SOP to a multi-level design showed that our method is also better than previous methods in terms of the area-delay product. In our future work, we will develop a synthesis method that directly targets at multi-level stochastic circuits.

## Acknowledgment

This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61472243 and 61204042.

## References

[1] B. R. Gaines, "Stochastic computing systems," in *Advances in information systems science*. Springer, 1969, pp. 37–172.

[2] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 93–105, 2011.

[3] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Design Automation Conference*, 2013, pp. 136:1–136:6.

[4] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. D. Riedel, "Computation on stochastic bit streams: Digital image processing case studies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 3, pp. 449–462, 2014.

[5] M. H. Najafi and M. E. Salehi, "A fast fault-tolerant architecture for Sauvola local image thresholding algorithm using stochastic computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 808–812, 2016.

[6] H. Ichihara, T. Sugino, S. Ishii, T. Iwagaki, and T. Inoue, "Compact and accurate digital filters based on stochastic computing," *accepted by IEEE Transactions on Emerging Topics in Computing*, 2016.

[7] Y. Liu and K. K. Parhi, "Architectures for recursive digital filters using stochastic computing," *IEEE Transactions on Signal Processing*, vol. 64, no. 14, pp. 3705–3718, 2015.

[8] S. S. Tehrani, S. Mannor, and W. J. Gross, "Fully parallel stochastic LDPC decoders," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5692–5703, 2008.

[9] S. S. Tehrani, A. Naderi, G.-A. Kamendje, S. Hemati, S. Mannor, and W. J. Gross, "Majority-based tracking forecast memories for stochastic LDPC decoding," *IEEE Transactions on Signal Processing*, vol. 58, no. 9, pp. 4883–4896, 2010.

[10] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rosselló, "A new stochastic computing methodology for efficient neural network implementation," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 3, pp. 551–564, 2016.

[11] B. Li, M. H. Najafi, and D. J. Lilja, "Using stochastic computing to reduce the hardware requirements for a restricted Boltzmann machine classifier," in *International Symposium on FPGA*, 2016, pp. 36–41.

[12] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Design Automation Confererence*, 2016, pp. 124:1–124:6.

[13] B. D. Brown and H. C. Card, "Stochastic neural computation I: Computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, 2001.

[14] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *International Conference on Computer-Aided Design*, 2012, pp. 480–487.
[15] A. Alaghi and J. Hayes, "STRAUSS: Spectral transform use in stochastic circuit synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1770–1783, 2015.
[16] Z. Zhao and W. Qian, "A general design of stochastic circuit and its synthesis," in *Design, Automation & Test in Europe*, 2015, pp. 1467–1472.
[17] K. Parhi and Y. Liu, "Computing arithmetic functions using stochastic logic by series expansion," *accepted by IEEE Transactions on Emerging Topics in Computing*, 2016.
[18] Y. Liu and K. K. Parhi, "Computing polynomials using unipolar stochastic logic," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, pp. 42:1–42:30, 2017.
[19] T.-H. Chen and J. P. Hayes, "Equivalence among stochastic logic circuits and its application," in *Design Automation Conference*, 2015, pp. 131:1–131:6.
[20] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Design Automation Conference*, 2008, pp. 648–653.
[21] X. Peng and W. Qian, "A branch-and-bound-based minterm assignment algorithm for synthesizing stochastic circuit," in *International Workshop on Logic and Synthesis*, 2016, pp. 155–162.
[22] W. Qian and M. D. Riedel, "The synthesis of stochastic logic to perform multivariate polynomial arithmetic," in *International Workshop on Logic and Synthesis*, 2008, pp. 79–86.
[23] P. Li, W. Qian, M. D. Riedel, K. Bazargan, and D. J. Lilja, "The synthesis of linear finite state machine-based stochastic computational elements," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 757–762.
[24] N. Saraf and K. Bazargan, "Polynomial arithmetic using sequential stochastic logic," in *Great Lakes Symposium on VLSI*, 2016, pp. 245–250.
[25] A. Alaghi and J. P. Hayes, "A spectral transform approach to stochastic circuits," in *International Conference on Computer Design*, 2012, pp. 315–321.
[26] S. A. Salehi, Y. Liu, M. D. Riedel, and K. K. Parhi, "Computing polynomials with positive coefficients using stochastic logic by double-NAND expansion," in *Great Lakes Symposium on VLSI*, 2017, pp. 471–474.
[27] S. W. Golomb, *Shift Register Sequences, Revised Ed.* Aegean Park Press, 1981.
[28] J. Berchtold and A. Bowyer, "Robust arithmetic for multivariate Bernstein-form polynomials," *Computer-Aided Design*, vol. 32, no. 11, pp. 681–689, 2000.
[29] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
[30] D. Baneres, J. Cortadella, and M. Kishinevsky, "A recursive paradigm to solve boolean relations," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 512–527, 2009.
[31] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 5, pp. 727–750, 1987.
[32] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," *URL: http://www.eecs.berkeley.edu/˜alanmi/abc*, 2007.

**Xuesong Peng** is a master student in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. He received his B.S. degree from the same Institute. He is interested in electronic design automation.

**Weikang Qian** is an assistant professor in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. He received his Ph.D. degree in Electrical Engineering at the University of Minnesota in 2011 and his B.Eng. degree in Automation at Tsinghua University in 2006. His main research interests include electronic design automation and digital design for emerging technologies. His research works were nominated for the Best Paper Awards at the 2009 International Conference on Computer-Aided Design (ICCAD) and the 2016 International Workshop on Logic and Synthesis (IWLS).