A Branch-and-Bound-Based Minterm Assignment Algorithm for Synthesizing Stochastic Circuit

Xuesong Peng and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {sayson, qianwk}@sjtu.edu.cn

Abstract—**Stochastic computing (SC) is an unconventional paradigm which performs computation on stochastic bit streams using ordinary digital circuits. In SC, a real number is encoded by a stochastic bit stream, where the value is the probability of ones in the stream. SC can perform complex arithmetic computation with simple circuits. It also has strong tolerance to bit flip errors. In a prior work, researchers have proposed a general design of stochastic circuit and a synthesis method. However, the synthesis method cannot produce a circuit with minimal area. In this paper, we propose an improved synthesis method that applies a branch-and-bound algorithm to search for the optimal minterm assignment. We also introduce a few techniques to speed up the synthesis procedure with only small quality loss. Experimental results showed that the new synthesis method produces smaller circuits.**

Keywords—stochastic computing; stochastic circuit synthesis; minterm assignment

I. Introduction

Stochastic computing (SC) is an alternative to the conventional computing paradigm based on binary radix encoding. In SC, digital circuits are still used to perform computation. However, their inputs are stochastic bit streams [1]. Each stochastic bit stream encodes a value equal to the probability of a 1 in the stream. For example, the stream *A* shown in Fig. 1 encodes the value 0.75.

One major advantage of SC is that it allows complex arithmetic computation to be realized by a very simple circuit. Fig. 1 shows that arithmetic multiplication can be realized by an AND gate, since for an AND gate, the probability of obtaining a 1 in the output bit stream is equal to the product of the probabilities of obtaining a 1 in the input bit streams.

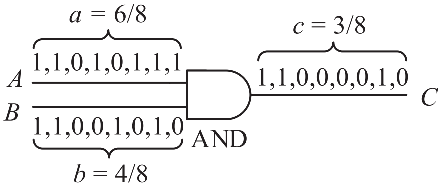

Fig. 1. An AND gate performs multiplication on real values encoded by stochastic bit streams.

Since all the bits in the stream have equal weight and a long bit stream is usually used to encode a value, a single bit flip occurring anywhere in the bit stream only causes very small change to the encoded value. Therefore, SC is highly tolerant to bit flip errors [2].

Given its advantages of low hardware cost and strong error tolerance, SC has been used in a number of applications, including image processing [3], decoding of modern error-correcting codes [4], and artificial neural networks [5].

In early days, various elementary computing units in SC were proposed, such as multiplier, scaled adder, divider, and squaring unit [6]. These units were designed manually and can only perform a limited types of computations.

In order to apply SC to a broad range of target computations, several methods to synthesize stochastic circuits have been proposed recently. The works [2,7,8] focused on synthesizing reconfigurable stochastic circuits. In [2], the authors proposed a method based on Bernstein polynomial expansion to synthesize combinational logic-based stochastic circuits. In [7] and [8], the authors studied the form of the computation realized by SC using sequential circuits and proposed methods to synthesize such designs. The works [9−11] focused on synthesizing fixed stochastic circuits, which take less area than reconfigurable ones. In [9], the authors demonstrated a fundamental relation between stochastic circuits and spectral transform. Based on this, they proposed a general approach to synthesize stochastic circuits. In [10], the authors found that different Boolean functions could compute the same arithmetic function in SC and proposed the concept of stochastic equivalence class. They proposed a method to search for the optimal Boolean function within an equivalence class. However, their method can only be applied to synthesize multi-linear polynomials. In [11], the authors introduced a general combinational circuit for SC and analyzed its computation. They further proposed a method to synthesize low-cost fixed stochastic circuit to realize a general polynomial.

The study in [11] reveals that in SC, there are a large number of different Boolean functions that realize the same target arithmetic function. Of course, the circuits for different Boolean functions have different costs. In previous work [11], a greedy method was used to find a circuit with low area cost. However, given the extremely large search space, the greedy strategy, although very fast, may not give a minimal solution. In this work, we address this problem by applying a branch-and-bound-based algorithm to extensively search for a Boolean function that will lead to a circuit with low cost. Our approach constructs a function by iteratively adding cubes into the on-set of the Boolean function. The optimal set of cubes to be added is determined through the search process. To improve the runtime, we also introduce a few speed-up techniques.

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

- We introduce a new method that iteratively selects cubes to form a Boolean function that realizes the target computation in SC.
- We develop a branch-and-bound algorithm to search for the optimal set of cubes to be added.
- We propose several speed-up techniques which prune unpromising branches and significantly improve the runtime of the algorithm.

The rest of the paper is organized as follows. In Section II, we give the background on the general design proposed in [11] and illustrate the previous synthesis method. We also present the logic synthesis problem for stochastic computing. In Section III, we present the new algorithm. In Section IV, we discuss several speed-up techniques. In Section V, we show the experimental results. Finally, we conclude the paper in Section VI.

II. Background on Synthesizing Stochastic Circuits

In this section, we give the background on the general form of the stochastic circuit proposed in [11] and discuss the previous method to synthesize 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. The General Form and Its Computation

The general form of a stochastic circuit is shown in Fig. 2. The circuit is a combinational circuit. It computes an arithmetic function $f(x_1,\dots,x_n)$, which is encoded by the output bit stream. It has n inputs $X_1,\dots,X_n$, which are supplied with variable probabilities $x_1,\dots,x_n$, respectively. In order to offer freedom for realizing different functions, the circuit has m extra inputs $Y_1,\dots,Y_m$, each supplied with a constant probability of 0.5. They can be easily obtained by a linear feedback shift register (LFSR). The value of m affects the quantization error and is chosen according to the accuracy requirement. The large the value m is, the smaller the quantization error.

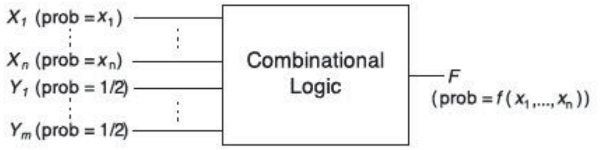

Fig. 2. General form of a stochastic circuit [11].

The study in [11] shows that the general design computes a type of function in the form

$$f(x_1,\dots,x_n)=\sum_{\substack{(a_1,\dots,a_n)\\ \in\{0,1\}^n}}\frac{g(a_1,\dots,a_n)}{2^m}\prod_{j=1}^{n}x_j^{a_j}(1-x_j)^{1-a_j},\ (1)$$

where $0\le g(a_1,\dots,a_n)\le 2^m$ is an integer. If the combinational circuit realizes a Boolean function $B(X_1,\dots,X_n,Y_1,\dots,Y_m)$, then the value $g(a_1,\dots,a_n)$ is equal to the number of vectors $(b_1,\dots,b_m)\in\{0,1\}^m$ such that $B(a_1,\dots,a_n,b_1,\dots,b_m)=1$.

Example 1

Suppose the Boolean function of the combinational circuit in Fig. 2 is $B(X_1,X_2,Y_1,Y_2)=X_1Y_1+X_2Y_2$. Then $B(1,1,Y_1,Y_2)=Y_1+Y_2$. Since there are three vectors $(b_1,b_2)\in\{0,1\}^2$ making $B(1,1,b_1,b_2)$=1, the value $g(1,1)=3$. Similarly, we can derive $g(0,0)=0$, $g(0,1)=2$, and $g(1,0)=2$. Since $m=2$, according to Eq. (1), the output function is

$$f(x_1,x_2)=\frac{1}{2}(1-x_1)x_2+\frac{1}{2}x_1(1-x_2)+\frac{3}{4}x_1x_2.\ (2)\ \square$$

The function of the form shown in Eq. (1) is called a *binary combination polynomial (BCP)* [11]. If we expand a BCP, we can obtain a *multi-linear polynomial (MLP)* of the following form

$$f(x_1,\dots,x_n)=\sum_{(a_1,\dots,a_n)\in\{0,1\}^n}\frac{c(a_1,\dots,a_n)}{2^m}\prod_{j=1}^{n}x_j^{a_j},$$

where $c(a_1,\dots,a_n)$'s are integers. The degree of each variable in an MLP is at most 1. For example, expanding Eq. (2), we can obtain an MLP

$$f(x_1,x_2)=\frac{1}{2}x_1+\frac{1}{2}x_2-\frac{1}{4}x_1x_2.$$

B. Synthesis of General Function

Given a target function, a procedure was proposed in [11] to synthesize a stochastic circuit of the general form to realize that function. We use an example to illustrate the procedure. Since the computation realized by a general-form stochastic circuit is a polynomial, the target function will be first approximated as a polynomial.

Now suppose the polynomial is $f=\frac{1}{4}x_1^2+\frac{1}{2}x_2$. Next, it will be transformed into an MLP. This is achieved by introducing two new variables $x_{1,1}$ and $x_{1,2}$ with their values both set as x_1. The MLP obtained is

$$f=\frac{1}{4}x_{1,1}x_{1,2}+\frac{1}{2}x_2.$$

The next step is to map the MLP into a BCP. By a procedure shown in [11], the result is

$$f=\frac{1}{2}(1-x_{1,1})(1-x_{1,2})x_2+\frac{1}{2}(1-x_{1,1})x_{1,2}x_2$$
$$+\frac{1}{2}x_{1,1}(1-x_{1,2})x_2+\frac{1}{4}x_{1,1}x_{1,2}(1-x_2)+\frac{3}{4}x_{1,1}x_{1,2}x_2.\ (3)$$

Assume that the number of Y-variables is $m=2$ and the Boolean function is $B(X_{1,1},X_{1,2},X_2,Y_1,Y_2)$. Comparing Eq. (3) with Eq. (1), we can obtain that the Boolean function should satisfy that

$$g(0,0,0)=0, g(0,0,1)=2, g(0,1,0)=0, g(0,1,1)=2,$$
$$g(1,0,0)=0, g(1,0,1)=2, g(1,1,0)=1, g(1,1,1)=3.$$

However, since $x_{1,1}=x_{1,2}=x_1$, the terms $(1-x_{1,1})x_{1,2}x_2$ and $x_{1,1}(1-x_{1,2})x_2$ are the same. Also, the terms $(1-x_{1,1})x_{1,2}(1-x_2)$ and $x_{1,1}(1-x_{1,2})(1-x_2)$ are the same. Therefore, the requirement for the Boolean function can be relaxed as follows

$$g(0,0,0)=0, g(0,0,1)=2, g(0,1,0)+g(1,0,0)=0,$$
$$g(0,1,1)+g(1,0,1)=4, g(1,1,0)=1, g(1,1,1)=3.\ (4)$$

In the general case, suppose the target polynomial has k variables $x_1, \dots, x_k$ and the degree of x_i is d_i, for $i = 1, \dots, k$. Define $n = \sum_{i=1}^{k} d_i$. To transform the original target into an MLP, we will introduce n new variables $x_{1,1}$, ..., x_{1,d_1}, ..., $x_{i,1}$, ..., x_{i,d_i}, ..., $x_{k,1}$, ..., x_{k,d_k}, with the values of $x_{i,1}, \dots, x_{i,d_i}$ all set to x_i. The BCP has 2^n product terms of the form

$$\prod_{i=1}^{k}\prod_{j=1}^{d_i} x_{i,j}^{a_{i,j}}\left(1 - x_{i,j}\right)^{1-a_{i,j}},$$

where $\left(a_{1,1}, \dots, a_{1,d_1}, \dots, a_{k,1}, \dots, a_{k,d_k}\right) \in \{0,1\}^n$. Each product term has a one-to-one correspondence to a vector $\left(a_{1,1}, \dots, a_{1,d_1}, \dots, a_{k,1}, \dots, a_{k,d_k}\right) \in \{0,1\}^n$. We call the vector *the characteristic vector* of the product term. We partition the set $\{0,1\}^n$ into $\prod_{i=1}^{k}(1 + d_i)$ *equivalence classes* $I(s_1, \dots, s_k)$, $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$, where

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

Under the condition that for all $1 \le i \le k$, $x_{i,1} = \cdots = x_{i,d_i} = x_i$, two product terms are the same if and only if their characteristic vectors belong to the same equivalent class. Therefore, to realize the target polynomial, we only require that the sum of the g values over all the vectors in an equivalence class is equal to a specific constant. Mathematically, the requirement is that for all $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$

$$\sum_{\substack{\left(a_{1,1}, \dots, a_{k,d_k}\right) \\ \in I(s_1, \dots, s_k)}} g\left(a_{1,1}, \dots, a_{k,d_k}\right) = G(s_1, \dots, s_k), \qquad (5)$$

where $0 \le G(s_1, \dots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ is a constant that can be derived by adding up the corresponding g values of an initial BCP transformed from the original target function.

The example shown before corresponds to a situation in which $k = 2, d_1 = 2$, and $d_2 = 1$. Then we have six equivalence classes

$I(0,0) = \{(0,0,0)\}, I(0,1) = \{(0,0,1)\}$,
$I(1,0) = \{(0,1,0), (1,0,0)\}, I(1,1) = \{(0,1,1), (1,0,1)\}$,
$I(2,0) = \{(1,1,0)\}, I(2,1) = \{(1,1,1)\}$.

Given the above equivalence classes, the requirement on the g values specified by Eq. (5) is same as Eq. (4) we derived before.

C. The Circuit Synthesis Problem

Eq. (5) shows a requirement on the Boolean function to realize the target polynomial. However, there are a large number of Boolean functions that can satisfy the requirement. In order to synthesize an optimal circuit, we need to find an optimal Boolean function that satisfies the requirement. For simplicity, we focus on two-level circuit in this work and we use the literal number of the sum-of-product (SOP) form as the cost measure. The optimization problem is stated as follows.

> Given an integer m and $\prod_{i=1}^{k}(1 + d_i)$ integers $G(0, \dots, 0)$, ... , $G(d_1, \dots, d_k)$ such that $0 \le G(s_1, \dots, s_k) \le 2^m \prod_{i=1}^{k} \binom{d_i}{s_i}$ for any $0 \le s_1 \le d_1, \dots, 0 \le s_k \le d_k$, determine an optimal Boolean function such that its g values satisfy Eq. (5).

The above problem has flexibility in determining the final Boolean function. However, it is different from the traditional logic minimization with don't cares or Boolean relation minimization problem [12]. The problem we consider here has a constraint on the number of input vectors belonging to a subset that make the function evaluate to 1. Thus, the determination of the output for an input vector will reduce the output choices of the other input vectors belonging to the same subset. In contrast, logic minimization with don't cares or Boolean relation minimization does not have that constraint. The determination of the output of an input vector does not reduce the output choices for the other input vectors. Therefore, solving the above problem requires a new method.

Suppose the Boolean function is $B(X_{1,1}, \dots, X_{1,d_1}, \dots, X_{k,1}, \dots, X_{k,d_k}, Y_1, \dots, Y_m)$. We represent it using a matrix, where the columns represent the X-variables and the rows represent the Y-variables. Both the columns and the rows are arranged in Gray code order. An example is shown in Fig. 3 for a case where $k = 1$, $d_1 = 3$, and $m = 2$.

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

Fig. 3. The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{X_1}\,\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$.

Using that matrix representation, the number $g\left(a_{1,1}, \dots, a_{k,d_k}\right)$ is equal to the number of ones in the column $a_{1,1} \dots a_{k,d_k}$. Then the optimization problem is to distribute $G(s_1, \dots, s_n)$ ones to columns corresponding to the vectors in the class $I(s_1, \dots, s_n)$ to achieve an optimal Boolean function. A method was proposed in the previous work [11] to find a good solution. It applies a greedy strategy to distribute the ones. Assume $l = \lfloor G(s_1, \dots, s_n)/2^m \rfloor$. Then the method sets the g values of the first l vectors in the class $I(s_1, \dots, s_n)$ as 2^m, the g value of the $(l+1)$-th vector as $(G(s_1, \dots, s_n) - 2^m l)$, and the g values of the remaining vectors as 0. The following example illustrates how the previous method works.

Example 2

Consider a case where $k = 1$, $d_1 = 3$, and $m = 2$. There are four equivalence classes for this case:

$I(0) = \{(0,0,0)\}, I(1) = \{(0,0,1), (0,1,0), (1,0,0)\}$,
$I(2) = \{(0,1,1), (1,0,1), (1,1,0)\}, I(3) = \{(1,1,1)\}$.

Assume the sums of g values over all the vectors in each equivalence class are $G(0) = 2$, $G(1) = 6$, $G(2) = 6$, and

$G(3) = 2$. For equivalence classes $I(0)$ and $I(3)$, each of them covers one column. We set $g(0,0,0) = 2$ and $g(1,1,1) = 2$. For equivalence classes $I(1)$ and $I(2)$, each of them covers three columns. Since $\lfloor G(1)/2^m \rfloor = 1$, we assign $g(0,0,1) = 4$, $g(0,1,0) = 2$, and $g(1,0,0) = 0$. Similarly, for class $I(2)$, we assign $g(0,1,1) = 4$, $g(1,1,0) = 2$, and $g(1,0,1) = 0$. The final assignment of the ones is shown in Fig. 3. The Boolean function is $B = \overline{X_1}\,\overline{Y_1} + X_2\overline{Y_1} + \overline{X_1}X_3$, which has 6 literals. □

However, the previous method may not give an optimal solution. For the case shown in Example 2, a better assignment is shown in Fig. 4, which gives a function $B = \overline{Y_1}$. In this work, we explore a better solution to the optimization problem.

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

Fig. 4. The matrix representation of the Boolean function $B(X_1, X_2, X_3, Y_1, Y_2) = \overline{Y_1}$.

III. The Proposed Algorithm

In this section, we present the new algorithm. For simplicity, we focus on univariate polynomials, i.e., $k = 1$. Our work can be extended to handle multivariate polynomials. The only difference is that there are more equivalence classes for multivariate cases. For univariate case, we have $n = d_1$ and we assume the n X inputs are $X_1, \ldots, X_n$.

The basic approach we use to construct an optimal solution is to add cubes one by one into the on-set of the Boolean function. Although the previous work also uses this strategy, it only adds cubes which cover minterms in the same equivalence class. In contrast, our method also adds cubes across different equivalence classes.

A. Preliminaries

Before presenting the details, we first introduce a few notations and definitions. We use $M(a_1, \ldots, a_n, b_1, \ldots, b_m)$ to denote the minterm corresponding to an input vector $(a_1, \ldots a_n, b_1, \ldots, b_m) \in \{0,1\}^{n+m}$. We say a minterm $M(a_1, \ldots, a_n, b_1, \ldots, b_m)$ is in an equivalence class $I(i)$ $(0 \le i \le n)$ if $(a_1, \ldots, a_n) \in I(i)$.

We use a vector $(v_0, \ldots, v_n)$ to represent numbers of unassigned minterms for $(n + 1)$ equivalent classes. We call such a vector *problem vector*. Initially, the problem vector is equal to $(G(0), \ldots, G(n))$, given by the problem specification. With cubes 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 of length $(n + 1)$. It is formed by the numbers of minterms of the cube in each equivalence class. We call such a vector *cube vector*. In order to distinguish it from the problem vector, we represent the cube vector using square brackets. For example, assume that $n = 2$ and $m = 1$. The cube X_1 contains four minterms $X_1X_2\overline{Y_1}$, $X_1\overline{X_2}\,\overline{Y_1}$, $X_1X_2Y_1$ and $X_1\overline{X_2}Y_1$, as shown in Fig. 5(a). The minterms $X_1\overline{X_2}\,\overline{Y_1}$ and $X_1\overline{X_2}Y_1$ are in the equivalence class $I(1)$ and the minterms $X_1X_2\overline{Y_1}$ and $X_1X_2Y_1$ are in the equivalence class $I(2)$. There are no minterms of the cube X_1 in the equivalence class $I(0)$. Therefore, the vector of the cube X_1 is [0,2,2]. Note that although each cube has a unique cube vector, a cube vector may correspond to a number of different cubes. For example, the cube X_2 has the same cube vector as the cube X_1, as shown in Fig. 5.

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

(a) Cube X_1

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

(b) Cube X_2

Fig. 5. 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 manipulate on the vector, it is important to know the valid form of a cube vector. We have the following claim on this.

Theorem 1

A cube vector is of the form $\left[0, \ldots, 0, 2^l\binom{r}{0}, 2^l\binom{r}{1}, \ldots, 2^l\binom{r}{r}, 0, \ldots, 0\right]$, where $0 \le r \le n$ and $0 \le l \le m$ are the numbers of the missing X-variables and missing Y-variables in the cube, respectively. The cube vector has t zeros at the beginning and $(n - t - r)$ zeros at the end, where $0 \le t \le n - r$ is equal to the number of uncomplemented X-variables in the cube and $(n - t - r)$ is equal to the number of complemented X-variables in the cube. □

Proof: Consider the matrix representation of the cube. Since there are l missing Y-variables in the cube, the cube covers 2^l rows and all the covered rows have the same pattern. Note that each covered row is also a cube, which contains all the m Y-variables. Therefore, we only need to show that for such a cube, its cube vector is of the form $\left[0, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0\right]$.

We consider the X-variables of the cube. Suppose that there are t uncomplemented X-variables and r missing X-variables in the cube. Then, the cube has $(n - t - r)$ complemented X-variables. The cube covers 2^r minterms, among which $\binom{r}{i}$ minterms are in the equivalence class $I(t + i)$, for $i = 0, \ldots, r$. For any $0 \le j < t$ or $t + r < j \le n$, there are no minterms of the cube in the equivalence class $I(j)$. Therefore, the cube vector is of the form $\left[0, \ldots, 0, \binom{r}{0}, \binom{r}{1}, \ldots, \binom{r}{r}, 0, \ldots, 0\right]$, in which there are t zeros at the beginning and $(n - t - r)$ zeros at the end. □

Example 3

Assume that $n = 3$ and $m = 2$. Then, the cube X_1Y_1 contains 8 minterms $X_1\overline{X_2}\,\overline{X_3}\,Y_1\overline{Y_2}$, $X_1\overline{X_2}\,\overline{X_3}\,Y_1Y_2$, $X_1\overline{X_2}\,X_3\,Y_1\overline{Y_2}$, $X_1\overline{X_2}\,X_3\,Y_1Y_2$, $X_1X_2\,\overline{X_3}\,Y_1\overline{Y_2}$, $X_1X_2\,\overline{X_3}\,Y_1Y_2$, $X_1X_2X_3Y_1\overline{Y_2}$, $X_1X_2X_3Y_1Y_2$. Its cube vector is $[0,2,4,2] =$

$\left[0,2\binom{2}{0},2\binom{2}{1},2\binom{2}{2}\right]$. For this cube vector, $l = 1$ is equal to the number of missing Y-variables and $r = 2$ is equal to the number of missing X-variables. The number of zeros at the beginning is 1, which is equal to the number of uncomplemented X-variables in the cube. The number of zeros at the end is 0, which is equal to the number of complemented X-variables in the cube. □

B. *The Basic Idea*

As mentioned at the beginning of this section, our approach iteratively adds cubes into the on-set of the Boolean function. Each time a cube is added, some entries in the problem vector will be reduced. When the problem vector becomes zero, the Boolean function is constructed.

Generally, a cube added later may intersect with a cube added previously. However, in our approach, we restrict that a cube added later should be disjoint to any cubes added before. For simplicity, we call this restriction *disjointness constraint*. Although this restriction may cause some quality loss, it has two benefits. First, it makes the counting of minterms easy, because we do not need to consider the overlapped minterms. With a cube satisfying the disjointness constraint added, the problem vector can be easily updated by subtracting the cube vector from the original problem vector. Second, the constraint 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 $\overline{X_1}X_2$. 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 [13]. Thus, the final result is a set of non-disjoint cubes corresponding to a minimum SOP expression.

In each iteration, when picking a cube, we also require that each entry in the cube vector of the cube is no larger than the corresponding entry in the current problem vector. For simplicity, we call this constraint *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 in choosing the cube to be added: we choose the largest cube among all valid cubes. The reasons for this are 1) in two-level logic synthesis, larger cubes have fewer literals and 2) with the largest cubes added, the problem vector is reduced most. The details of how we choose the largest valid cube will be discussed in Section III-C. The procedure of choosing the largest valid cube involves obtaining a cube corresponding to the cube vector, which will be discussed in Section III-D. Since at each iteration, there may exist more than one largest valid cube for the current problem setup, we actually apply a branch-and-bound algorithm to find the optimal solution, which will be discussed in Section III-E.

C. *Selecting the Largest Valid Cube*

Suppose that at the beginning of one iteration, the problem vector is $(v_0, \dots v_n)$. Let s be the sum of all the entries in the problem vector, i.e., $s = \sum_{i=0}^{n} v_i$. Assume $q = \lfloor \log_2 s \rfloor$. Since the largest valid cube satisfies the capacity constraint, it contains at most 2^q minterms. Our method to find the largest valid cube first checks whether there exists a valid cube with 2^q minterm.

According to Theorem 1, the cube vector should be in the form of $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$, where $0 \le r \le n$ and $0 \le l \le m$. Furthermore, since the cube contains 2^q minterms, we require that $l + r = q$. We will examine all cube vectors that satisfy the above two requirements and keep those which also satisfy the capacity constraint. Then, for each kept cube vector, we will check whether it has a corresponding cube that satisfies the disjointness constraint. The details of how to check the existence of such a cube will be discussed in Section III-D. If such a cube exists, it is a largest valid cube.

Example 4

Suppose $n = 2$, $m = 2$, and we are given an initial problem vector of $(2,5,2)$. The sum of all the entries in the problem vector is 9. Thus, the largest valid cube has at most 8 minterms. We first check whether there exists any valid cube with 8 minterms. This type of cubes should be in the form of $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$ with $0 \le r \le 2$, $0 \le l \le 2$, and $l + r = 3$. Given the constraint, we have either $l = 2$ and $r = 1$, or $l = 1$ and $r = 2$. Thus, the possible cube vectors are $[0,4,4]$, $[4,4,0]$, and $[2,4,2]$. Among these three cube vectors, only the cube vector $[2,4,2]$ satisfies the capacity constraint. Then, we will further check whether it has a corresponding cube satisfying the disjointness constraint. Since no cubes have been added yet, we can find a valid cube for the cube vector $[2,4,2]$, for example, the cube Y_1. This cube is one largest valid cube. □

In some situations, there may not exist a valid cube with 2^q minterms because either the capacity constraint or the disjointness constraint is violated. The following is an example.

Example 5

Suppose $n = 2$, $m = 3$, and we are given an initial problem vector of $(1,3,7)$. The sum of all the entries in the problem vector is 11. Thus, the largest valid cube has at most 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 $[2,4,2]$. However, none of these cube vectors satisfy the capacity constraint. Therefore, we cannot find a valid cube with 8 minterms. □

If there exists no valid cube with 2^q minterms, then we 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 we are able to find a valid cube with 2^i minterms for some $0 \le i \le q$. Then, that cube is the largest valid cube. Since in the worst case, we can always find a minterm that is valid, the procedure guarantees to terminate at some point.

However, in general cases, the largest valid cube is not unique. This is due to the existence of more than one largest cube vector that satisfies the capacity constraint and the existence of more than one cube for a cube vector.

Example 6

Suppose $n = 2$, $m = 3$, and we are given an initial problem vector of $(4,8,3)$. The largest possible cube has 8 minterms. Among all cube vectors of 8 minterms, three satisfy the capacity constraint: [0,8,0], [4,4,0], and [2,4,2]. Furthermore, there exists more than one cube that satisfies the disjointness constraint for each of the three cube vectors. For example, for the cube vector [0,8,0], it corresponds to cubes $X_1\overline{X_2}$ and $\overline{X_1}X_2$, which satisfy the disjoint constraint. Therefore, there exist more than one largest valid cubes for this case. □

When there are multiple choices of the largest valid cubes, we want to evaluate all of them and choose the best one. For this purpose, we apply a branch-and-bound algorithm to find an optimal Boolean function. The details of it will be discussed in Section III-E.

D. Obtainting Cubes for a Cube Vector

In this section, we discuss one important procedure in selecting the largest valid cube: obtaining cubes for a given cube vector that satisfies the disjointness constraint. Since a cube is composed of X-variables and Y-variables, the procedure is divided into two parts: determining the X-variables and determining the Y-variables.

The X-variables are determined based on the form of the cube vector. As shown in Theorem 1, if the vector is of the form $\left[0,\dots,0,2^l\binom{r}{0},2^l\binom{r}{1},\dots,2^l\binom{r}{r},0,\dots,0\right]$ where there are t zeros at the beginning and $(n-t-r)$ zeros at the end, then the set of X-variables is composed of t uncomplemented X-variables and $(n-t-r)$ complemented X-variables. For example, if $n=3$ and the cube vector is of the form [0,4,4,0], then the possible X-variable cubes are $X_1\overline{X_2}$, $X_1\overline{X_3}$, $X_2\overline{X_1}$, $X_2\overline{X_3}$, $X_3\overline{X_1}$, and $X_3\overline{X_2}$.

Next, for each set of possible X-variables, we will further determine all sets of Y-variables so that the cube formed by these X-variables and Y-variables satisfies the disjointness constraint. According to Theorem 1, the set of Y-variables we need to pick consists of $(m-l)$ Y-variables. To obtain all valid sets of Y-variables, we can simply enumerate all cubes consisting of $(m-l)$ Y-variables and keep those when combined with the X-variable cube do not overlap with the current Boolean function. However, we could find a large number of valid Y-variable cubes, which increases the number of largest valid cubes. In order to reduce the choices, in our implementation, we enumerate all cubes with $(m-l)$ Y-variables in the Gray code order and keep the first valid Y-variable cube for each set of possible X-variables.

E. Branch-and-Bound Algorithm

As we mentioned before, in each iteration, there may exist more than one largest valid cube. If this happens, it is hard to decide which one will minimize the literal number of the final Boolean function. Therefore, we apply a branch-and-bound algorithm to evaluate all possible cube choices. An example of the search tree is shown in Fig. 6. Each leaf of the search tree corresponds to a final solution, represented by a set of cubes. Each internal node stores a partial solution composed of a set of cubes added and the remaining problem vector. The root is the initial problem vector. At each internal node, the multiple choices of the largest valid cubes for the current problem vector lead to multiple branches from the node.

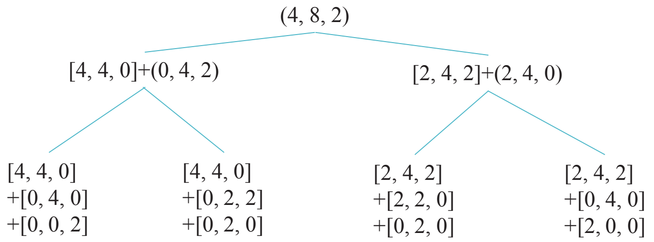

Fig. 6. An illustration of the solution tree for the problem with the problem vector $(4,8,2)$ and $m=2$. Note that for simplicity, we use a cube vector to represent a cube and we only show a partial tree.

Algorithm 1. Branch-and-bound algorithm to find optimal function.

```
inputs: problem vector $v = (G_0, \dots, G_n)$ and an integer $m$
outputs: the set of cubes of the final Boolean function $B$
initialize a node $N$: $N.vector \Leftarrow v$; $N.cube\_set \Leftarrow \emptyset$;
initialize the optimal literal number $n_o \Leftarrow \infty$;
initialize the optimal cube set $S_o \Leftarrow \emptyset$;
push the node $N$ into an empty stack $Stk$;
while $Stk$ is not empty do
  pop a node $N$ out of $Stk$;
  find a list $L$ of largest valid cubes for $N.vector$, $N.cube\_set$,
  and $m$;
  for each cube $C$ in the list $L$ do
    if $lit\_count(N.cube\_set \cup C) < n_o$ then
      $N_{new}.vector \Leftarrow N.vector - vector(C)$;
      $N_{new}.cube\_set \Leftarrow N.cube\_set \cup C$;
      if $N_{new}.vector = 0$ then // reach a leaf
        $n_o \Leftarrow lit\_count(N_{new}.cube\_set)$;
        $S_o \Leftarrow N_{new}.cube\_set$;
      else
        push the node $N_{new}$ into $Stk$;
      end if
    end if
  end for
end while
return $S_o$;
```

In order to apply a brand-and-bound algorithm, we need a lower bound on the candidate solutions from a branch. We choose the lower bound as the minimum literal number for the set of cubes that forms a partial solution at a branch. For example, for the branch $[2,4,2]+(2,4,0)$ shown in Fig. 6, its lower bound is the minimum literal number for the cube with the cube vector [2,4,2]. Strictly speaking, the minimum literal number for the set of chosen cubes at a branch may not be the lower bound for that branch, because with more cubes determined later, it is possible to reduce the literal count due to cube expansion and redundant cube removal. However, since the cubes selected later are no larger than any of the cubes already chosen, it is more likely that with more cubes selected, the literal count will increase. Thus, we use the proposed method to obtain the lower bound. A branch will be pruned if the lower bound for the branch is larger than or equal to the minimum literal count for the best solution obtained so far. In practice, the exact minimum literal number for a set of cubes is computationally expensive to obtain. Instead, we call the powerful two-level logic optimization tool ESPRESSO [13] to estimate the minimum value. Algorithm 1 summarizes the proposed branch-and-bound algorithm to find an optimal solution. Note that we explore the solution tree using the depth-first traversal.

IV. Speed-up Techniques

Although the branch-and-bound algorithm deletes some unpromising branches, there are still too many branches to process as the degree of the polynomial increases, which increases the runtime considerably. However, there are numerous branches unnecessary to process, either because they are unpromising or because they produce the same results. In this section, we present several techniques to speed up the algorithm with only small quality loss.

A. Removing Branches with Duplicated Cube Sets

For a node in the search 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 cube may not be 2^q. Example 5 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 branch-and-bound algorithm, the order that these cubes are added can produce different branches. Nevertheless, in most cases, different orders will finally lead to the same results.

Example 7

Suppose $n = 2$, $m = 3$, and the initial problem vector is $(1,6,2)$. We cannot extract a valid cube of size 8 from the initial problem vector. As a result, the largest valid cube is of size 4. Its cube vector is either [1,2,1] or [0,4,0]. With the original algorithm, if the first cube selected is of the cube vector [1,2,1], then the second cube selected will be of the cube vector [0,4,0]. On the other hand, if the first cube selected is of the cube vector [0,4,0], then the second cube selected will be of the cube vector [1,2,1]. These two branches from the root node will produce the same results. □

Those branches with the same set of cubes as a branch explored before are unnecessary to be explored again. To remove them, we keep track of the sets of cube vectors we have already examined. If the set of the cube vectors at the current branch has been examined before, the branch will be pruned.

B. Bounding by the Optimal Cost at Each Level

In the original algorithm, a branch is pruned only when its lower bound exceeds the value of the optimal full solution known so far. In practice, given that each time we always add a largest valid cube, it is very likely that for any level i in the search tree, the cost of the partial solution at level i in a branch that will be pruned later is larger than the cost of the optimal partial solution at level i. In other words, only those branches with costs close to the optimal partial solution at each level are promising in leading to the optimal full solution. Therefore, we propose another speed-up technique which prunes branches based on the cost of the optimal partial solution at each level. With this technique, we can find and prune many unpromising branches earlier. However, the proposed method is just a heuristic. In order to reduce the quality loss caused by applying this heuristic, we choose the bound at each level as the cost of the optimal partial solution at the current level multiplied by a constant $m_l > 1$. We will only delete those branches whose costs exceed the bound. In real implementation, since we traverse the solution tree in a depth-first way, the optimal partial solution is obtained among all the explored nodes at the current level.

C. Limiting Update Count and Explored Node Number

The previous two speed-up techniques focus on eliminating unpromising branches. However, for some extreme cases, the numbers of nodes explored could still be very large. In order to further reduce the runtime for these extreme cases, we impose limits on the update count and the number of explored nodes.

Our algorithm will update the optimal solution if the current solution is no worse than the optimal one recorded. As a result, each update will either improve the result or leave it unchanged. Experimental results showed that with more updates, the improvement will gradually reduce. Therefore, we consider the solution to be optimal enough after a specific number of updates. Thus, we set a limit on the update number and terminate the algorithm once the limit is reached. From our experimental results, we set this limit as 3. The quality loss is negligible.

Even though limiting the updating number can further improve the runtime for some extreme cases, there are still some cases for which a large number of nodes are explored between two consecutive updates. In our experiment, there is a recorded case for which after the second update, the algorithm processed 16463 other nodes to reach the third update. It took about 57 minutes to explore these nodes, but no improvement was made for the third update. Therefore, we also set a limit on the number of explored nodes. The algorithm records the number of nodes explored. Once the initial solution has been found, the number of nodes explored will be compared against the limit and the algorithm will terminate once the limit is reached. In our experiment, the limit is often set from 15 to 30 for $3 \leq n \leq 7$ and $3 \leq m \leq 7$, or larger if needed. With a larger limit, we can achieve a better solution.

V. Experiment Results

In this section, we show the experimental results of the proposed algorithm. All the experiments were conducted on a desktop with 3.20GHz Intel(R) Core(TM) i5-4570 CPU and 16.0 GB RAM. ESPRESSO is used to evaluate the literal count [13].

We applied the proposed branch-and-bound algorithm with the speed-up technique to univariate polynomials with $3 \leq n \leq 7$ and $3 \leq m \leq 7$. For each pair of n and m, we generated 50 random cases and obtained the average result. Table I shows for each pair of n and m, the average percentage of literal count reduction by the proposed algorithm over the method in [11], the percentage of improved or unchanged cases among all 50 cases, and the average runtime in seconds of the proposed algorithm. The literal reduction percentage, the percentage of improved and unchanged cases, and the runtime are shown in the first row, the second row, and the third row of each cell. For example, for $n = 4$ and $m = 4$, the proposed algorithm saves 13% literal count on average. 100% of the 50 cases have their literal counts reduced or unchanged. The average runtime is 1.42s.

TABLE I. THE AVERAGE PERCENTAGE OF LITERAL COUNT REDUCTION BY THE PROPOSED ALGORITHM OVER THE PREVIOUS METHOD [11] (IN THE FIRST ROW OF EACH CELL), THE PRECENTAGE OF IMPROVED AND UNCHANGED CASES (IN THE SECOND ROW OF EACH CELL), AND THE AVERAGE RUNTIME OF THE PROPOSED ALGORITHM (IN THE THIRD ROW OF EACH CELL) FOR DIFFERENT PAIRS OF n AND m.

$m\backslash n$	3	4	5	6	7
3	6% 100% 0.44s	12% 90% 0.60s	19% 94% 1.60s	18% 88% 2.56s	26% 100% 5.20s
4	3% 98% 0.60s	13% 100% 1.42s	16% 88% 2.46s	22% 92% 4.20s	29% 94% 7.48s
5	2% 92% 0.88s	13% 100% 1.14s	18% 92% 2.92s	18% 84% 8.74s	26% 88% 17.1s
6	2% 92% 1.34s	12% 96% 3.90s	15% 86% 7.04s	18% 88% 16.6s	23% 90% 42.6s
7	2% 88% 2.46s	10% 94% 8.54s	14% 86% 16.6s	17% 90% 39.2s	22% 90% 116s

It can be seen that in the average sense, the proposed algorithm reduces the literal count compared to the previous method. When n is small, the literal count reduction is small 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 29%. For each pair of n and m, at least 84% of cases have their literal counts improved or unchanged. For some pairs of n and m, all 50 cases have their literal counts improved or unchanged. With the increase of n and m, the runtime also increases, which is due to the growth of the search space. Notice that the runtime of the previous method is negligible compared to ours, due to its greedy nature. However, the runtime of our algorithm is still affordable. In situations where better circuit quality is pursued, our method gives a better solution under a reasonable amount of runtime.

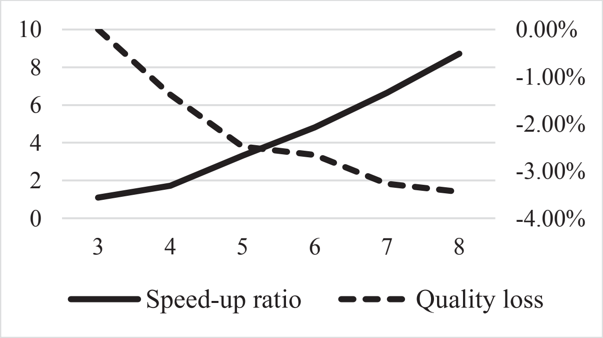

Fig. 7. Comparison between the branch-and-bound algorithm without acceleration and the accelrated algorithm for $n = 3$ and $3 \leq m \leq 8$.

We also compared the proposed accelerated algorithm to the branch-and-bound algorithm without using the speed-up techniques. Due to the inefficiency of the algorithm without acceleration, the comparison was only done for polynomials of degree $n = 3$ and $3 \leq m \leq 8$. Fig. 7 plots the speed-up ratio (shown in solid line, y-axis on the left) and the quality loss (shown in dashed line, y-axis on the right) of the accelerated algorithm for different m values. For the quality loss, the more negative the value is, the more loss the accelerated algorithm has. We can see from Fig. 7 that as the problem instance grows, more runtime can be saved through the speed-up techniques. However, the quality loss also increases. Nevertheless, the quality loss is small. Indeed, in terms of the absolute value, the average quality loss is smaller than one literal. Thus, the speed-up techniques have a negligible impact on the quality.

VI. CONCLUSION

In this work, we proposed a search-based method for synthesizing stochastic circuits. The synthesis problem we considered here is different from the traditional logic synthesis problem in that there exist many different Boolean functions to realize a target computation. We proposed a branch-and-bound algorithm to systematically explore the solution space. A final solution is obtained by adding a series of cubes to the on-set of the Boolean function. We also provided several speed-up techniques. The experimental results showed that our algorithm produces smaller circuits than a previous greedy approach, especially when the target polynomial has a high degree.

ACKNOWLEDGMENT

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

REFERENCES

[1] B. Gaines, “Stochastic computing systems,” in *Advances in Information Systems Science*. Plenum, 1969, vol. 2, ch. 2, pp. 37–172.

[2] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. 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. Hayes, “Stochastic circuits for real-time imageprocessing applications,” in *DAC*, 2013, pp. 1–6.

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

[5] B. Brown and H. Card, “Stochastic neural computation II: Soft competitive learning,” *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, 2001.

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

[7] P. Li, W. Qian, M. Riedel, K. Bazargan, and D. Lilja, “The synthesis of linear finite state machine-based stochastic computational elements,” in *ASPDAC*, 2012, pp. 757–762.

[8] P. Li, D. Lilja, W. Qian, K. Bazargan, and M. Riedel, “The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic,” in *ICCAD*, 2012, pp. 480–487.

[9] A. Alaghi and J. Hayes, “A spectral transform approach to stochastic circuits,” in *ICCD*, 2012, pp. 315–321.

[10] T.-H. Chen and J. Hayes, “Equivalence among stochastic logic circuits and its application,” in *DAC*, 2015, pp. 131-136.

[11] Z. Zhao and W. Qian, “A general design of stochastic circuit and its synthesis,” in *DATE*, 2015, pp. 1467-1472.

[12] D. Baneres, J. Cortadella, and M. Kishinevsky, “A recursive paradigm to solve Boolean relations,” in *DAC*, 2004, pp. 416-421.

[13] R. Rudell, “Multiple-valued logic minimization for PLA synthesis,” *Technical Report, University of. California, Electronics Research Laboratory, Berkeley,* 1986.