Approximate Disjoint Bi-decomposition and Its Application to Approximate Logic Synthesis

Yue Yao, Shuyang Huang, Chen Wang, Yi Wu and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {patrickyao, King_hsy, wangchen_2011, eejessie, qianwk}@sjtu.edu.cn

***Abstract*— Approximate computing is an emerging design paradigm targeting at error-tolerant applications. The area, delay, and power consumption of a circuit can be improved by sacrificing a reasonable amount of accuracy. Approximate logic synthesis (ALS) aims at automatically synthesizing an approximate circuit for a given target circuit. In this paper, we propose to approximate a target function by a maximally disjoint bi-decomposable function, which can significantly reduce the implementation cost. We propose novel techniques to effectively generate such an approximation with low error rate. We further integrate this approximation technique into a systematic ALS flow. Experiment results showed the effectiveness of our proposed ALS flow in producing approximate circuits with reduced areas.**

***Index Terms*—approximate logic synthesis, approximate computing, inexact circuit, Boolean function decomposition**

I. Introduction

As the level of integration for VLSI designs increases, the power consumption of a design also dramatically grows. This calls for technologies that simplify the circuit without affecting their effectiveness in their application context. Meanwhile, many recent applications tolerate errors by their nature. Given this context, a new computing paradigm called *approximate computing* [1][2] was proposed. This paradigm sacrifices accuracy in hope of generating a much simpler circuit so that both its area and power consumption can be reduced. Many studies employed this idea at different design levels, including algorithm, architecture, logic, and transistor level, and demonstrated the effectiveness of this design paradigm [2].

One challenging problem in designing approximate circuits is to find the optimal circuit that approximates the target design. In previous works, researchers have either manually derived an approximate circuit [3][4][5][6], or developed algorithms to synthesize one. The latter option is known as *approximate logic synthesis* (ALS) [7][8]. This strategy aims at designing algorithms that automatically simplify a given target circuit by introducing a small amount of approximation to the original function. There are some ALS approaches for two-level circuits [7][9] and some for multilevel circuits [10][11][12].

In traditional logic synthesis, functional decomposition is a widely-used technique for synthesizing circuits with smaller areas [13][14][15]. A general decomposition method decomposes a large function into a combination of smaller functions, and hence could reduce the implementation cost. A special type of decomposition is the disjoint bi-decomposition (DBD), which represents a function $f(X)$ as $D(g_1(X_1), g_2(X_2))$, where X is the input set of f, D, g_1, and g_2 are Boolean functions, and X_1 and X_2 form a bipartition of the set X [16]. If the functions g_1 and g_2 can be recursively disjoint bi-decomposed, then the Boolean function f can be maximally disjoint bi-decomposed (MDBDed). As a result, it can be realized by a tree of logic gates, which significantly reduces its implementation cost. Unfortunately, most Boolean functions cannot be MDBDed. However, in the approximate computing context, since error is allowed, it is possible to approximate the target function by another one that can be MDBDed. Previous works proposed a few efficient bi-decomposition algorithm, including the BDD-based algorithm in [17] and the spectral method in [18]. However, these methods can only produce a decomposition when the function is bi-decomposable. In contrast, our work tries to produce a bi-decomposition that is closet to the target function even if the function is not bi-decomposable. Toward this end, we first study the necessary and sufficient condition for a function to be disjoint bi-decomposed. Then, based on that result, we develop a method to find a disjoint bi-decomposable function that is closest to the target function. Based on this technique, we further develop a method that can find a Boolean function that can be maximally disjoint bi-decomposed and is closest to the target function. Finally, a systematic ALS flow incorporating the proposed technique is proposed. The experimental results demonstrated that our ALS flow can produce approximate circuits with significantly reduced areas.

To evaluate the error resulted from approximation, two major metrics are commonly used. The first is *error rate*, which is the percentage of input patterns that result in an inaccurate output. The second is *error magnitude*, which is the maximal numerical difference of the output by treating all the output bits encoding a binary number. In this study, we choose error rate instead of error magnitude as the error constraint, since for many logic circuits, the error magnitude may not be well-defined.

The rest of the paper is organized as follows. In Section II, we introduce the concept of disjoint bi-decomposition. In Section III, we elaborate our proposed approach to perform approximate disjoint bi-decomposition. In Section IV, we describe our proposed method to perform approximate maximum disjoint bi-decomposition (AMDBD). In Section V, we present the ALS flow that exploits the AMDBD technique. In Section VI, we show the experiment results. Finally, in Section VII, we conclude the paper.

II. Disjoint Bi-decomposition

In this work, we propose techniques to perform approximate disjoint bi-decomposition. First, we introduce a few definitions.

Definition 1: Let the set of input variables be $X = \{x_1, x_2, \ldots, x_n\}$. If two non-empty subsets X_1 and X_2 satisfy that $X_1 \cup X_2 = X$ and $X_1 \cap X_2 = \emptyset$, then (X_1, X_2) is a **bipartition** of X [19]. □

Definition 2: Let $F(X) : \mathbf{B}^n \mapsto \mathbf{B}$ be a Boolean function and (X_1, X_2) be a bipartition of X. The **decomposition chart (DC)** of F over the biparition (X_1, X_2) is a table of $2^{|X_1|}$ columns and $2^{|X_2|}$ rows, with columns labeled by all the binary combinations of the variables in X_1, rows by all the binary combinations of the variables in X_2, and the intersection of each row and each column by the value of F on the corresponding combination of inputs specified by the column and the row [19]. □

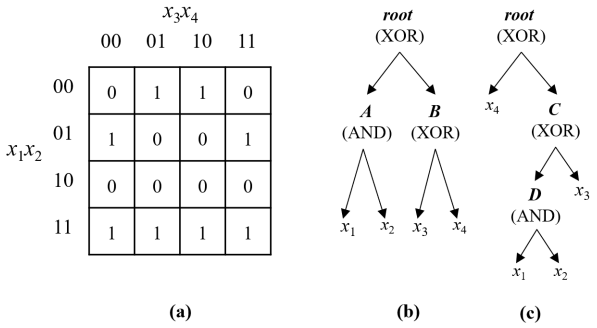

Fig. 1: Illustration of a few concepts. (a): The decomposition chart of a Boolean function $f(x_1, \ldots, x_4)$ over the bipartition $X_1 = \{x_3, x_4\}$ and $X_2 = \{x_1, x_2\}$. (b): An MDBD tree. (c): Another MDBD tree, which implements the same function as the one in (b).

Example 1: Consider a function $f(x_1, x_2, x_3, x_4) = x_1'x_2'x_3'x_4 + x_1'x_2'x_3x_4' + x_1'x_2x_3'x_4' + x_1'x_2x_3x_4 + x_1x_2$ and a bipartition $X_1 = \{x_3, x_4\}$ and $X_2 = \{x_1, x_2\}$. Then, the DC of f over the bipartition (X_1, X_2) is shown in Fig. 1. □

The concept of **disjoint bi-decomposition** is defined as follows [16].

Definition 3: If a Boolean function $F(X) : \mathbf{B}^n \mapsto \mathbf{B}$ can be represented as $F(X) = D(g_1(X_1), g_2(X_2))$, where (X_1, X_2) is a bipartition of the input set X and g_1, g_2, and D are three Boolean functions, then this representation of F is referred to as a **disjoint bi-decomposition (DBD)** of F. The Boolean function F is **disjoint bi-decomposable**. □

Based on the definition of disjoint bi-decomposition, we further define maximally disjoint bi-decomposable function, which is defined recursively as follows.

Definition 4: A Boolean function $F(X) : \mathbf{B}^n \mapsto \mathbf{B}$ is **maximally disjoint bi-decomposable (MDBD)**, if

1) $|X| = 1$, or
2) there exists a DBD of F, $D(g_1(X_1), g_2(X_2))$, such that both g_1 and g_2 are MDBD. □

By its definition, an MDBD function can be implemented as a tree of $(n-1)$ 2-input gates. We define such a tree as a **maximum disjoint bi-decomposition tree (MDBD tree)**. Fig. 1(b) shows an example of an MDBD tree. Note that two different MDBD trees may implement the same MDBD function. As an example, the function implemented by the tree in Fig. 1(b) is

$$F_1(x_1, x_2, x_3, x_4) = (x_1x_2) \oplus (x_3 \oplus x_4),$$

while the one implemented by tree in Fig. 1(c) is

$$F_2(x_1, x_2, x_3, x_4) = (x_3 \oplus (x_1x_2)) \oplus x_4.$$

These two trees implement the same Boolean function.

Clearly, among all Boolean functions that **depend on** n variables, an MDBD function can be implemented with the minimum number (i.e., $n-1$) of 2-input gates. Our proposed approximate logic synthesis approach exploits this fact. In order to maximize the area reduction while minimizing the impact on accuracy, our proposed approach tries to replace the Boolean function of a local circuit by an MDBD function that is closest to the Boolean function.

In order to find the closest MDBD function, we first study when a function has a disjoint bi-decomposition. We have the following claim, which is related to a result on decomposition proposed by Ashenhurst [13].

Theorem 1: A Boolean function $F(X) : \mathbf{B}^n \mapsto \mathbf{B}$ has a disjoint bi-decomposition $F(X) = D(g_1(X_1), g_2(X_2))$ if and only if the DC of F over the bipartition (X_1, X_2) has at most **two** distinct types of rows belonging to the following four categories:

1) a pattern of all 0's;
2) a pattern of all 1's;
3) a fixed pattern of both 0's and 1's;
4) the complement of the pattern in Category 3. □

The proof of the "only if" part of Theorem 1 is omitted due to space limit. We prove the "if" part of the theorem, which also demonstrates how to obtain a disjoint bi-decomposition $D(g_1(X_1), g_2(X_2))$ when the condition is satisfied.

If the condition is held, then there are six cases on the rows of the DC. Examples of these six cases are shown in Fig. 2(a)–(f), respectively. These examples are 4-input functions with $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4)$. Next, we discuss each case.

1) The case where all the rows of the DC of F fall into a single category that is Category 1 (or 2). Fig. 2(a) shows an example for this case, where all the rows of the DC are all-0. Then, F is a constant 0 (or 1). Thus, D is a constant function of 0 (or 1). g_1 and g_2 can be arbitrary functions on X_1 and X_2, respectively. For the example in Fig. 2(a), $D(g_1, g_2) = 0$.
2) The case where all the rows of the DC of F fall into a single category that is Category 3 (or 4). Fig. 2(b) shows an example for this case, where all the rows of the DC are of the same pattern that are neither all-0 nor all-1. Clearly, F does not depend on X_2. Thus, $D(g_1, g_2) = g_1$, $g_1(X_1)$ is given by the pattern of an arbitrary row, and g_2 can be an arbitrary function. For the example in Fig. 2(b), we have $D(g_1, g_2) = g_1$ and $g_1(X_1) = \overline{x_1}x_2$.
3) The case where all the rows of the DC of F fall into 2 categories that are Categories 1 and 2. Fig. 2(c) shows an example for this case. Clearly, F does not depend on X_1. Thus, $D(g_1, g_2) = g_2$, $g_2(X_2)$ is given by the pattern of an arbitrary column, and g_1 can be an arbitrary function. For the example in Fig. 2(c), we have $D(g_1, g_2) = g_2$ and $g_2(X_2) = x_3\overline{x_4}$.
4) The case where all the rows of the DC of F fall into 2 categories that are Categories 1 and 3 (or 4). Fig. 2(d) shows an example for this case. For this case, D is

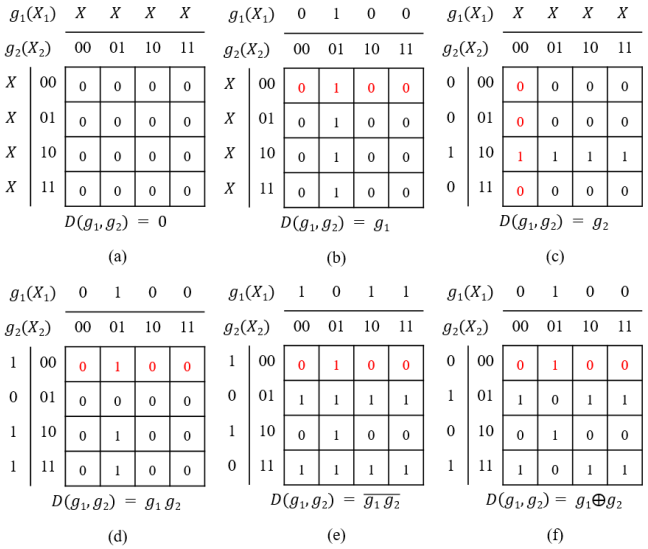

Fig. 2: The six cases on the rows of the decomposition chart satisfying the condition of Theorem 1.

an **AND** function. g_1 is given by the row pattern of Category 3 (or 4). g_2 is determined as follows. For the row corresponding to an input combination of the variables in X_2, if the row is all-0, then the value of g_2 for that input combination is 0; otherwise, the value is 1. For the example in Fig. 2(d), we have $D(g_1, g_2) = g_1 g_2$, $g_1(X_1) = \overline{x_1} x_2$, and $g_2(X_2) = x_3 + \overline{x_4}$.

5) The case where all the rows of the DC of F fall into 2 categories that are Categories 2 and 3 (or 4). Fig. 2(e) shows an example for this case. For this case, D is an **NAND** function. g_1 is given by the negation of the row pattern of Category 3 (or 4). g_2 is determined as follows. For the row corresponding to an input combination of the variables in X_2, if the row is all-1, then the value of g_2 for that input combination is 0; otherwise, the value is 1. For the example in Fig. 2(e), we have $D(g_1, g_2) = \overline{g_1 g_2}$, $g_1(X_1) = x_1 + \overline{x_2}$, and $g_2(X_2) = \overline{x_4}$.
6) The case where all the rows of the DC of F fall into 2 categories that are Categories 3 and 4. Fig. 2(f) shows an example for this case. For this case, D is an **XOR** function. g_1 is given by the pattern of the first row in the DC. g_2 is determined as follows. For the row corresponding to an input combination of the variables in X_2, if the row is identical to the first row, then the value of g_2 for that input combination is 0; otherwise, the value is 1. For the example in Fig. 2(f), we have $D(g_1, g_2) = g_1 \oplus g_2$, $g_1(X_1) = \overline{x_1} x_2$, and $g_2(X_2) = x_4$.

In what follows, we refer to the above 6 cases as decomposition choices $1, 2, \ldots, 6$, respectively.

We further make the following definition.

Definition 5: A 2-input function $f(x_1, x_2)$ is **trivial** if f does not depend on both x_1 and x_2. A disjoint bi-decomposition $D(g_1, g_2)$ is **trivial** if D is a trivial 2-input function. □

Clearly, for the decomposition choices 1, 2, and 3, their DBDs are trivial, while for the decomposition choices 4, 5, and 6, their DBDs are non-trivial. By our decomposition method shown above, for a non-trivial DBD $D(g_1, g_2)$, D is one of AND, NAND, and XOR functions.

For an MDBD tree of an MDBD function f, by simplifying its internal trivial 2-input functions, we will obtain an MDBD tree such that all of its internal 2-input functions are non-trivial. We refer to such an MDBD tree as a **non-trivial MDBD tree** of the function f. We refer to the function of a non-trivial MDBD tree as a **non-trivial MDBD function**. Without loss of generality, in what follows, we assume that every 2-input function of a non-trivial MDBD tree is one of AND, NAND, and XOR functions.

III. Approximate Disjoint Bi-decomposition

As we mentioned in the previous section, in our proposed approximate logic synthesis approach, given an arbitrary local Boolean function, we will find an MDBD function that is closest to the Boolean function. By the recursive nature of an MDBD function, a key problem is to find a DBD that is closest to the given Boolean function. In this section, we present a method to solve this key problem.

A. Problem Formulation

The problem we solve in this section is formally stated as follows: Given a Boolean function $F(X)$ and a partition (X_1, X_2) of the input set X, find a DBD function $G(X) = D(g_1(X_1), g_2(X_2))$ such that the error rate of $G(X)$ over $F(X)$ is the smallest among all DBD functions of the form $D(g_1(X_1), g_2(X_2))$. The error rate of the function $G(X)$ over the function $F(X)$ is calculated as the sum of the occurrence probabilities of the input combinations X's that let $G(X) \neq F(X)$. We call the function $G(X)$ the **optimal approximate DBD** of the function $F(x)$.

As we will see later, we will apply our proposed approximate decomposition approach to local circuits of the given circuit. Thus, the input set X is a set of local inputs. In this work, the occurrence probability of each possible binary combination of the local inputs is obtained through logic simulation, assuming the uniform random distribution of the primary inputs of the circuit. With these probabilities known, we can evaluate the error rate of an approximate function.

By our proof of Theorem 1, we only need to consider 6 decomposition choices. To obtain the optimal approximate DBD, we first obtain the optimal approximate DBD for each specific decomposition choice. Then, the final optimal approximate DBD is given by the decomposition choice with the smallest error rate. Next, we discuss how to obtain the optimal approximate DBD for each specific decomposition choice.

B. Solutions for Decomposition Choices 1, 2, and 3

Obtaining the optimal approximate DBD for decomposition choices 1, 2, and 3 is simple. For choice 1, we compare the given function with the constant-0 and the constant-1 functions. We choose the one that minimizes the error rate as the optimal approximate DBD. For choice 2, as shown in Fig. 2(b), each column of the DC of the approximate function G should be either all-0 or all-1. In order to find the G that minimizes the error rate, For each column c of the DC of the given function F, we compare it with the all-0 column and the all-1 column, and pick the one that minimizes the error rate over c as the corresponding column pattern in the DC of G. Decomposition choice 3 is similar to choice 2 except that each row of the DC of the approximate function G should be either all-0 or all-1, as shown in Fig. 2(c). The optimal approximate DBD for choice 3 can be obtained in a similar way as choice 2.

C. Solutions for Decomposition Choices 4, 5, and 6

In order to determine the optimal approximate DBD for decomposition choices 4, 5, and 6, we need to solve two key problems, identifying a proper fixed row pattern and identifying the set of rows that should be assigned with this fixed pattern.

In what follows, we consider the decomposition choice 6, where D is an XOR function. The methods for handling decomposition choices 4 and 5 are similar and hence, omitted.

To help identify the optimal approximate DBD, we define a **probability matrix** K of the same size as the DC. Each entry in the probability matrix gives the occurrence probability of the input pattern corresponding to the combination of the column index and row index. If we flip the output of a given input pattern, the error rate is increased by the occurrence probability of the input pattern.

An example is given in Fig. 3. In this example, the function has 4 inputs. The input set X is bipartitioned as $X = (X_1, X_2)$, where $|X_1| = |X_2| = 2$. The matrix M gives the DC of the function, while the matrix K gives the probability matrix. For simplicity, the numbers in K are in percentage.

A solution for the decomposition choice 6 consists of 2 parts: a fixed pattern P as mentioned in Theorem 1 and a decision that for each row of the DC of G, whether it is set as the pattern P or its complement, $\overline{P}$. We represent the decision using a 0-1 vector PF of length $2^{|X_2|}$. The i-th entry in the vector is 0 (1) if and only if the i-th row of the DC of G is set as P ($\overline{P}$). Given the vectors P and PF, the approximate function G is fully determined. Therefore, identifying the optimal approximate

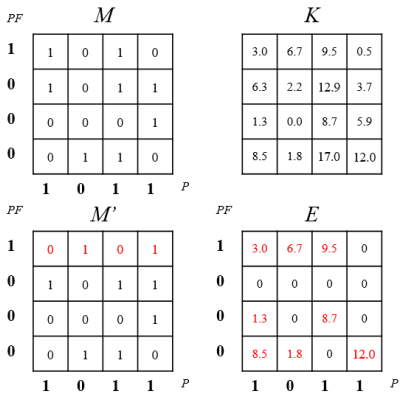

Fig. 3: An example showing the decomposition chart M of a given function, a solution (P, PF), the probability matrix K, the flipped decomposition chart M', and the error rate matrix E.

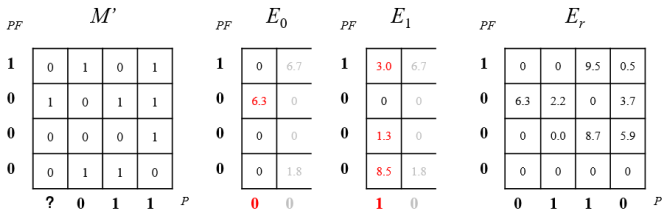

Fig. 4: Determining the first entry of the local minimum row vector. E_0 and E_1 are partially drawn due to space limit.

DBD is equivalent to identifying the vectors P and PF that minimize the error rate.

To evaluate the error rate of a given solution (P, PF), we first flip the rows in the DC of the function F that have a 1 in the corresponding entry in the vector PF. We refer to the result as the **flipped decomposition chart (flipped DC)**. For the example in Fig. 3, given the vector PF as shown on the left of the matrix M, we flip the first row of M and keep the remaining rows. We eventually obtain the matrix M' in Fig. 3, which is the flipped DC. Then, we compare each row of the matrix M' with the row pattern P, which is shown below the matrix M'. If an entry in an row is different from the corresponding entry in P, then an error occurs for the choice (P, PF) and the error rate is increased by the value of the corresponding entry in the matrix K. The matrix E highlights the entries in the matrix M that have an error and the associated error rates.

Before presenting our solution, we first make the following definition.

Definition 6: We say a solution (P, PF) is at a **local minimum** if we cannot improve the error rate by changing just one of PF and P, and it is at the **global minimum** if it has the minimum error rate among all possible solutions. □

Our ultimate goal is to identify a global minimum solution (P, PF). This is hard, since given that the vectors P and PF are of sizes $2^{|X_1|}$ and $2^{|X_2|}$, respectively, there are $2^{2^{|X_1|}}$ possible P's and $2^{2^{|X_2|}}$ possible PF's. Instead, we try to identify a local minimum solution.

We show how we obtain the local minimum solution using the example shown in Fig. 3. We start form a random initial solution (P, PF), where $P = (1, 0, 1, 1)$ and $PF = (1, 0, 0, 0)^T$, as shown in Fig. 3. We first improve the initial solution by fixing PF and finding a P that minimizes the error rate. The resultant row vector P is called a **local minimum row vector** for the column vector PF.

Given the fixed column vector PF, we construct a flipped DC for that PF to help identify the local minimum row vector

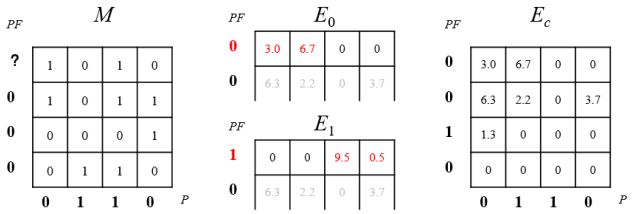

Fig. 5: Determining the first entry of the local minimum column vector. E_0 and E_1 are partially drawn due to space limit.

P. We find the local minimum row vector P entry by entry. Fig. 4 uses the first entry as an example. The matrix M' in the figure is the flipped DC for the current choice of $PF = (1, 0, 0, 0)^T$, which is same as the matrix M' in Fig. 3. We have 2 choices for the first entry, either 0 or 1. If we choose the entry as 0, then all the entries in the first column of the flipped DC that are 1 will induce errors. Otherwise, all the entries that are 0 will induce errors. We choose the value for the first entry that gives the smaller error rate. In our example, for the choice of 0 (1) for the first entry of P, the associated error rates for all the entries in the first column are shown in the first column of the matrix E_0 (E_1) in Fig. 4. Clearly, choosing the first entry as 0 gives a smaller total error rate. Thus, the first entry of P is determined as 0. In the case where there is a tie on the error rates between the choices of 0 and 1, we will keep the previous value in P.

For our example, iterating the above process for the remaining three entries of the row vector P, we finally determine the local minimal row vector P for the PF as $P = (0, 1, 1, 0)$ and the error rate for each entry is shown in the matrix E_r in Fig. 4. The error rate decreases to 36.8%.

Once we have determined the local minimum row vector P^* for a given PF, we will fix P as P^* and find the column vector PF that minimizes the error rate. The resultant column vector PF is called a **local minimum column vector** for the row vector P.

The local minimum column vector is also decided entry by entry. For each entry, if it is 0, then we will compare the row vector P with the corresponding row in the DC and obtain the error rate; otherwise, we will compare the complement of P with the corresponding row in the DC and obtain the error rate. We pick the choice for the entry that gives the smaller error rate.

Fig. 5 shows the example to further determine the local minimum column vector for the example in Fig. 4. In Fig. 5, the first row of the matrix E_0 (E_1) shows the error rate for each entry when the first entry of the column vector PF is chosen as 0 (1). Clearly, choosing the first entry as 0 gives a smaller total error rate. Thus, the first entry of PF is determined as 0. In the case where there is a tie on the error rates between the choices of 0 and 1, we will keep the previous value in PF. The remaining three entries of the column vector PF is determined in a similar way. The eventual local minimum column vector is $PF = (0, 0, 1, 0)^T$. The error rate for each entry is shown in the matrix E_c in Fig. 5. The error rate further decreases to 23.2%.

To obtain the final local minimum solution (P, PF), the above process of first solving the local minimum row vector for the latest PF and then solving the local minimum column vector for the latest P is repeated until the solution (P, PF) does not change over the previous iteration. Since each iteration reduces the error rate and the error rate is at least 0, the process will eventually terminate. Once the process terminates, it means

that the error rate of the final solution (P, PF) cannot be further improved by changing only one of P and PF. By Definition 6, the final solution is a local minimum.

For our example in Fig. 3, the proposed method reaches a local minimum after 2 iterations. The final local minimum solution has $P = (1, 0, 1, 0)$ and $PF = (0, 0, 1, 0)^T$, which are shown near the matrix E in Fig. 6. The matrix E gives the error rate for each entry for this solution. The total error rate is 14%, which is significantly improved from the original error rate of 51.5%.

Note that the proposed method for finding the local minimum solution can be easily adapted to handle decomposition choices 4 and 5.

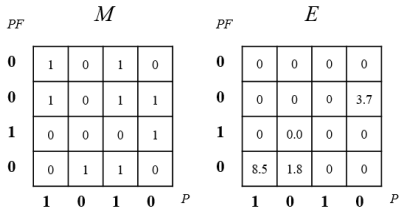

Fig. 6: The local minimum solution (P, PF) for the given decomposition chart M and the associated error matrix E.

IV. Approximate Maximum Disjoint Bi-Decomposition

In this section, we present our method for obtaining a maximally disjoint bi-decomposable function that **approximates** a given function. We refer to such a function as **approximate maximally disjoint bi-decomposable function**, or **approximate maximum disjoint bi-decomposition (AMDBD)** for short. Ideally, we want to find an optimal AMDBD that minimizes the error rate over the given function. Our proposed procedure relies on the method for finding the optimal approximate DBD for a given function and a given bipartition of the input set, which is described in Section III.

A. Basic Algorithm

Algorithm 1 shows the proposed function *ApxMaxDBD* for finding an optimal AMDBD for a given function. It is a recursive function. Line 2 handles the base case where the size of the input set is no more than 2. For such a case, the procedure simply returns the input function F. For a normal case, the procedure iterates over all the bipartitions (X_1, X_2)'s of the input set X (see Line 5). For each input bipartition (X_1, X_2), it calls the function *ApxDBDforPar*(F, X_1, X_2), which returns the optimal approximate DBD $D(apxG_1(X_1), apxG_2(X_2))$ for the function F and the given bipartition (X_1, X_2) (Line 6). The function implements the method described in Section III. Since $apxG_1$ may not be an MDBD function, in order to obtain the optimal AMDBD, we recursively call the function *ApxMaxDBD* itself on the functions $apxG_1$ to obtain the optimal AMDBD $optApxG_1$ for the function $apxG_1$ (Line 7). We do the same thing for the function $apxG_2$ (Line 8). After that, we combine the two-input function D, the function $optApxG_1$, and the function $optApxG_2$ together (Line 9). Clearly, the resultant function $apxF$ is an MDBD function. It is considered as the optimal AMDBD for the input function F for the bipartition (X_1, X_2), although, strictly speaking, it may not be the exact optimal one due to the local greedy choice of the two-input function D. Nevertheless, our experimental results in Section VI-A demonstrated that the final AMDBD returned by the function *ApxMaxDBD* is very close to the optimal result.

Algorithm 1: Function for obtaining the optimal approximate maximum disjoint bi-decomposition for a given function.

```
1  Function ApxMaxDBD(F, X)
       Input: a function F with input set X;
       Output: the optimal approximate maximum DBD
               optApxF for the function F;
2      if |X| ≤ 2 then return F ;
3      minER ⇐ +∞;
4      optApxF ⇐ 0 ;
5      for All parition (X1, X2) of the input set X do
6          (D, apxG1, apxG2) ⇐ ApxDBDforPar(F, X1, X2);
7          optApxG1 ⇐ ApxMaxDBD(apxG1, X1);
8          optApxG2 ⇐ ApxMaxDBD(apxG2, X2);
9          apxF ⇐ CombineFunc(D, optApxG1, optApxG2) ;
10         ER ⇐ ErrorRate(F, apxF) ;
11         if minER > ER then
12             minER ⇐ ER;
13             optApxF ⇐ apxF;
14         end
15     end
16     return optApxF;
```

The error rate of the function $apxF$ is computed by the function *ErrorRate* (Line 10) and if the error rate is smaller than any of the bipartitions that have been considered so far, the optimal AMDBD for the function F, $optApxF$, is updated (Lines 11–14). Finally, after all the bipartitions of the input set X are considered, $optApxF$ stores the best AMDBD among all the bipartitions and it is returned (Line 16).

It can be seen that for an input function with n variables, Algorithm 1 essentially performs a brute-force search over all MDBD trees with n leaves. Therefore, its runtime complexity is at least on the order of the number of structurally different full binary trees with n leaves, which is equal to the Catalan number $\Theta(\frac{4^n}{n^{3/2}})$ [20]. Thus, Algorithm 1 runs in exponential time, which is unaffordable for large n's. Also, as we mentioned in Section II, multiple different MDBD trees may have the same Boolean function. Therefore, even for a small n, the brute-force search of Algorithm 1 may undergo some redundant computation.

B. Speedup by Pre-computing

As we will show in Section V, in order to apply the proposed approximate maximum disjoint bi-decomposition technique to ALS, it is applied to local sub-circuits with limited number of inputs. Since we may consider many different local sub-circuits, Algorithm 1 will be called many times. Therefore, to improve the runtime for the entire ALS flow, instead of calling Algorithm 1 each time for a new function with n inputs, we can enumerate and store all the MDBD functions of n inputs first. Later on, for each function with n inputs, a simple scan over all the MDBD functions of n inputs can quickly return the optimal AMDBD for that function. For simplicity, we refer to this method as **enumeration-based method**.

In order to obtain all MDBD functions of n inputs, we first enumerate all the non-trivial MDBD trees of $0, 1, \ldots, n-1$ nodes. Note that in the enumeration, we also consider all the input permutations. Then, we obtain the functions of all these MDBD trees. Since multiple different MDBD trees may correspond to the same MDBD function, we only keep the unique functions. The numbers of different MDBD functions for $n = 3, 4, 5, 6$ inputs are 152, 2680, 68968, 2311640, respectively. However, when n is larger than 6, the number of different non-trivial MDBD functions becomes prohibitively large. Therefore, we only enumerate and store MDBD functions for n up to 6. Given a function of n inputs where $n \leq 6$,

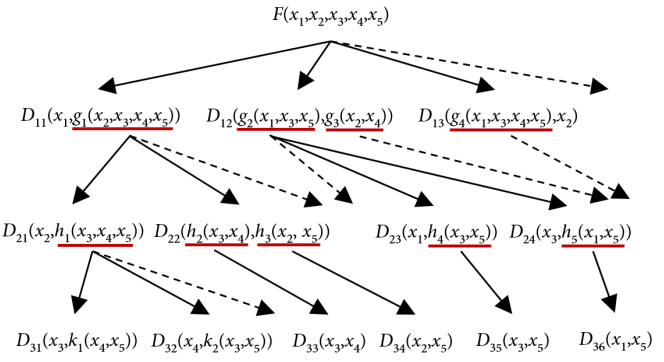

Fig. 7: A recursion tree generated by Algorithm 1. The dashed arrows represent those branches that are omitted due to space limit.

we find its optimal AMDBD through scanning all the MDBD functions of n inputs and selecting the one that minimizes the error rate.

C. Speedup by Branch-and-Bound and Branch Elimination

When the input size n is larger than 6, we cannot use the enumeration-based method proposed in the previous section. Therefore, we still need to apply Algorithm 1 to find an optimal AMDBD. We propose two techniques for speeding up Algorithm 1.

The first technique is to use branch-and-bound. During the execution of Algorithm 1, it keeps track of the current lowest error rate. Once we reach an MDBD in the recursion, the lowest error rate could be updated. At any point in the recursion, all the approximate DBDs on the path from the root to the current node in the recursion tree give a function F_c, which may not be an MDBD. For example, Fig. 7 shows part of the recursion tree generated by Algorithm 1. The approximate DBDs on the path from the root to the node $D_{21}(x_2, h_1(x_3, x_4, x_5))$ gives the function $D_{11}(x_1, D_{21}(x_2, h_1(x_3, x_4, x_5)))$. It may not be an MDBD, since $h_1(x_3, x_4, x_5)$ may not. To see whether there is a better MDBD that can be reached from the current node, we calculate the error rate of the function F_c over the given function F. We compare that error rate with the current lowest error rate. If it is smaller, then we continue the recursion from the current node. Otherwise, we stop the recursion. The reason is that further evolving from the current node may introduce more approximation and hence, increase the error rate. As a result, it is unlikely to lead to a better solution than the current best solution. For our example, if we find that the error rate of the function $F_c = D_{11}(x_1, D_{21}(x_2, h_1(x_3, x_4, x_5)))$ is larger than the current lowest error rate, then we will not further recursively call the function *ApxMaxDBD* on the Boolean function $h_1(x_3, x_4, x_5)$.

The branch-and-bound method may still be time-consuming when n becomes larger. Therefore, we propose the second technique, which is more aggressive in eliminating branches. The basic idea is that at each level of the recursion tree, we only keep k nodes with the smallest error rates, which are measured in a same way as the branch-and-bound method, and reach to the next level only through these nodes. To apply this technique, the solution space exploration is performed in a breadth-first manner. For the example shown in Fig. 7, suppose that $k = 2$ and the nodes $D_{12}(g_2(x_1, x_3, x_5), g_3(x_2, x_4))$ and $D_{13}(g_4(x_1, x_3, x_4, x_5), x_2)$ at the second level give the smallest error rates among all the nodes at that level. Then, we only reach to the next level from these two nodes. In our final implementation of Algorithm 1, we combine this branch elimination technique with the branch-and-bound technique. Also, at any point of the recursion, if the input size of the function is no more than 6, we will directly obtain the optimal AMDBD by the enumeration-based method.

V. Application to Approximate Logic Synthesis

In this section, we present how we apply the approximate maximum disjoint bi-decomposition (AMDBD) technique to approximate logic synthesis (ALS). The AMDBD technique can only handle a single-output function, but many digital circuits may have multiple outputs. Furthermore, this technique may not handle a circuit with a large number of inputs efficiently. In view of these limitations, we propose to apply this technique to local sub-circuits with a single output. Specifically, we choose to apply this technique to maximum fanout-free cones in a circuit. For each node v in a circuit, we can obtain a maximum fanin cone of the node so that all the nodes except v have their fanouts inside this cone. This cone is known as the **maximum fanout-free cones (MFFCs)** of the node v [21].

The flow chart of the proposed ALS method using the AMDBD technique is shown in Fig. 8. It takes as inputs a given circuit C and an error rate threshold r. Then, it enters into a loop of iteratively introducing approximation to the circuit. The loop terminates when the the error rate threshold is reached and then, the approximate circuit from the previous iteration is returned. In each iteration of the loop, we will select a set of MFFCs and replace them with their AMDBDs obtained by the proposed AMDBD technique.

One key problem is which set of MFFCs we should select and replace by their AMDBDs. We notice that there are much more small MFFCs with input sizes no more than 6 in a circuit than large MFFCs with input sizes more than 6. For these small MFFCs, we can easily find their optimal AMDBDs by the enumeration-based method described in Section IV-B. Furthermore, many of these AMDBDs only have small error rates. To take advantage of these facts and to overcome the long runtime disadvantage of finding AMDBDs for MFFCs with large input sizes, we design a two-phase selection strategy. Assume that the error rate of the approximate circuit of the current iteration is r_0. Then, the error rate margin we still have is $r - r_0$. The two-phase selection strategy will use part of the margin to accommodate small MFFCs and the remaining margin to accommodate large MFFCs. Specifically, we set another error rate threshold $r_1 = a(r - r_0)$, where $0 < a < 1$ is a constant. In the first phase, we obtain the AMDBDs for all the small MFFCs. Then, we select them in the increasing order of error rates until the threshold r_1 is reached. In the second phase, we select a number of large MFFCs until the remaining error rate margin $r - r_0 - r_1$ is reached. This is achieved through an inner loop. In each iteration of the inner loop, we just select one MFFC. To achieve this, we randomly select a fixed number l of large MFFCs as candidates and obtain their AMDBDs. Then, we use the ratio of the area saving over the error rate as a score and choose the one with the highest score among all the candidates.

VI. Experiment Results

In this section, we present experimental results on the proposed methods. We implemented the proposed methods in C++. All the experiments were conducted on a laptop computer with a 2.7GHz CPU with 16GB memory.

A. Optimality of the AMDBD Algorithm

In this experiment, we studied the optimality of Algorithm 1. The algorithm we tested is the basic version without applying any speedup techniques. We compared the result returned

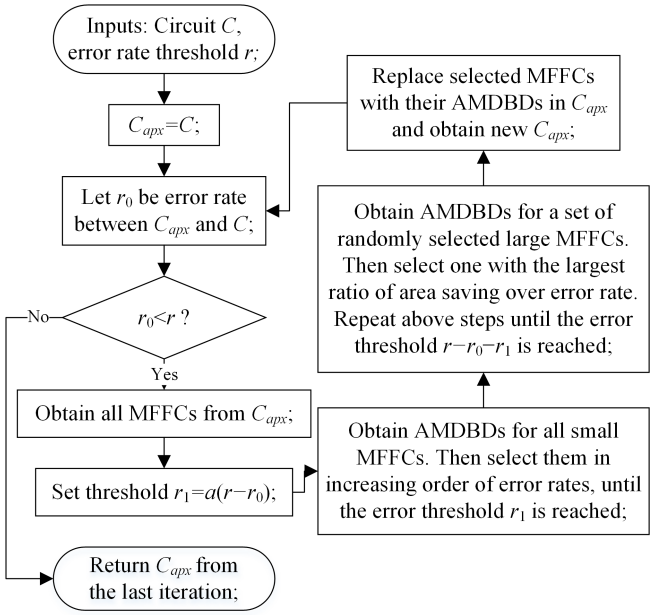

Fig. 8: Flow chart of the proposed ALS method.

by the algorithm against the optimal AMDBD returned by the enumeration-based method. Since the enumeration-based method works for functions with input size $n \leq 6$, we chose functions with $n = 3, \ldots, 6$ inputs as the input functions. For input sizes $n = 3$ and 4, we chose all possible functions with n inputs as the test cases. For input sizes $n = 5$ and 6, we randomly chose 10000 functions with n inputs as the test cases. For each test case, we counted the number of erroneous input vectors in the optimal AMDBD returned by the enumeration-based method. Then, for each $n = 3, \ldots, 6$, we average the numbers of erroneous input vectors over all the test cases of input size n. We did the same thing for Algorithm 1. For both methods, we also obtained the maximum number of erroneous input vectors among all test cases with n inputs, for $n = 3, \ldots, 6$. The average and maximum numbers are shown in Table I. From the table, we can see that both the average and the maximum numbers of erroneous input vectors are very close for these two methods. This indicates that Algorithm 1 can always find a close-to-optimal solution.

TABLE I: Comparing Algorithm 1 with the enumeration-based method.

Method	Enumeration		Algorithm 1	
n	**Avg. #**[1]	**Worst #**[2]	**Avg. #**[1]	**Worst #**[2]
3	0.406	1	0.406	1
4	1.78	4	1.79	4
5	5.478	8	5.58	9
6	14.5684	18	14.9535	19

[1] "Avg. #" is the average number of erroneous inputs vectors over all test cases.
[2] "Worst. #" is the maximum number of erroneous input vectors among all test cases.

B. Performance of Approximate Logic Synthesis

In this section, we show the experimental results on the proposed ALS method based on the AMDBD technique. Table II lists the test circuits used in the experiment, which include several circuits from the MCNC benchmark [22] and several arithmetic circuits synthesized by Synopsys Design Compiler [23].

TABLE II: Benchmark circuits.

Name	**# I/Os**	**# nodes**	**Area**
C880	60/26	316	607
C1908	33/25	364	906
ALU4	14/8	1451	2783
C7552	205/107	1521	3034
C3540	50/22	928	1810
APEX2	39/3	1720	3197
EX1010	10/10	1882	3321
SPLA	16/46	4066	7565

To measure the area of a circuit, we applied logic synthesis tool ABC [24] to perform technology mapping using the MCNC generic standard cell library [22]. In order to eliminate dangling nodes and constant nodes produced during the execution of our algorithm, we applied the *sweep* command in ABC before mapping. To be fair, we did the same to the original circuit. We measured the error rate of a circuit by randomly generating 10000 primary input (PI) vectors, assuming that all the input vectors appear with equal probabilities. Our method ignores the timing constraints in approximating the circuits. However, our experimental results showed that each approximate circuit produced by our method has a smaller delay than the corresponding input circuit.

We tested two methods in our experiment. The first method, called **basic method**, applies the basic version of Algorithm 1 to obtain the AMDBDs during the proposed ALS flow. The second method, called **accelerated method**, applies all the proposed speedup techniques to Algorithm 1 to obtain the AMDBDs. We allowed Algorithm 1 to approximate MFFCs with at most 11 inputs in the circuit. We chose the parameter k that determines the number of nodes kept in each level of the recursion tree as 3.

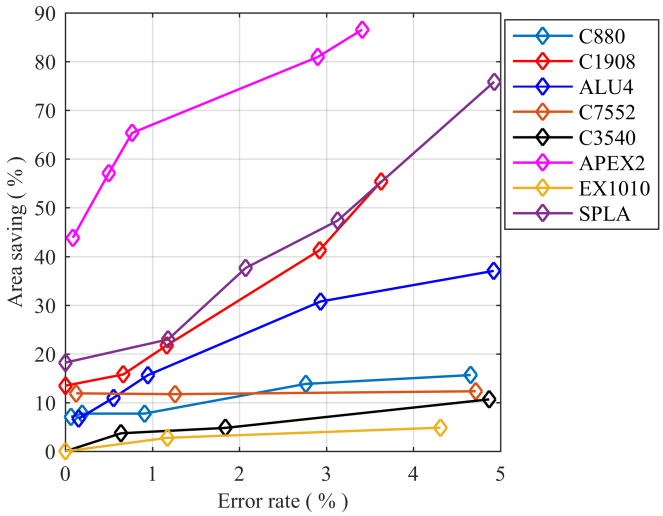

Fig. 9: Area saving versus error rate for different benchmarks by applying the accelerated method .

We chose five error rate thresholds: 0.2%, 0.5%, 1%, 3% and 5%. Fig. 9 shows the area saving versus error rate for all the benchmarks by applying the accelerated method. From Fig. 9, we can see that for all the benchmarks, the area saving increases (at least slightly) as the error rate threshold increases. In most cases, given an $x\%$ error rate, our ALS method could reduce the area by more than $2x\%$. For several cases such as C1908, APEX2, and SPLA, more than $15x\%$ area could be reduced.

Notice that for some circuits, the area saving is not zero when the error rate is 0. This is because our algorithm uses simulation to determine the error rate, which guides the approximation. Consequently, for an MFFC, an input pattern may never appear in the simulation. If the AMDBD for that MFFC has a different output than the MFFC for this input pattern, the introduced error rate is 0. It should be noted that such an input pattern may indeed occur in the circuit and hence, replacing the MFFC by the AMDBD could cause some error. However, since the input pattern do not occur in our random simulation with a large number of input vectors, we believe that the actual error rate may be negligible.

TABLE III: Area savings and runtimes of the basic method and the accelerated method.

	Accelerated Method		**Basic Method**	
Name	**Area Saving**	**Runtime(s)**	**Area Saving**	**Runtime(s)**
C880	10.4%	22	10.8%	479
C1908	29.5%	67	31.0%	2655
ALU4	20.2%	91	/	> 3600
C7552	12.0%	140	/	> 3600
C3540	6.4%	199	/	> 3600
APEX2	66.8%	104	/	> 3600
EX1010	3.8%	189	5.5%	3321
SPLA	40.4%	89	41.1%	2982

In order to demonstrate the advantages of the accelerated method over the basic method, we compared these two methods on their area savings and runtimes. Table III shows the results. The "Area Saving" column and the "Runtime" column give the average area saving and the average runtime, respectively, over the 5 different error rate thresholds for each benchmark. Since the basic method has a large runtime complexity, for circuits ALU4, C7552, C3540, APEX2, it cannot finish within a reasonable amount of time (1 hour in our experiment). However, the accelerated method could finish in less than 200 seconds for all the benchmarks. This demonstrates the effectiveness of the accelerated method. For the circuits C880, C1980, and SPLA, the quality loss of the accelerated method compared to the basic method is negligible.

Finally, we compared our accelerated method with the "multi-selection" method proposed in [10], which, to our best knowledge, is the state-of-the-art method for ALS under the error rate constraint. With the same settings, we compared the average area savings over the 5 error rate thresholds for these two methods. The results are listed in Table IV. From the table, we conclude that our method achieved comparable results as the state-of-the-art method. For three circuits ALU4, C7752, and C3540, our method can further improve the area saving by at least 30%, 70%, and 70%, respectively, compared to the method in [10].

TABLE IV: Comparison between our method and the previous method [10] on area saving.

Circuit	**C880**	**C1908**	**ALU4**	**C7552**	**C3540**
Our method	11%	30%	21%	12%	7%
Method in [10]	12%	37%	16%	7%	4%

VII. Conclusion

In this work, we proposed a novel technique to perform approximate disjoint bi-decomposition, which approximates a function by transforming it into a maximally disjoint bi-decomposable form. To reduce the long runtime of the basic approach, several techniques were proposed to accelerate the search of a good candidate approximation with low error rate. We applied this accelerated approach to approximate logic synthesis. Experimental results demonstrated the effectiveness of the proposed method in synthesizing approximate circuits.

Acknowledgments

This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61574089.

References

[1] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.
[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.
[3] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *International Symposium on Integrated Circuits*, 2009, pp. 69–72.
[4] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *International Conference on VLSI Design*, 2011, pp. 346–351.
[5] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," in *International Conference of Electron Devices and Solid-State Circuits*, 2010, pp. 1–4.
[6] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *International Conference on Computer-Aided Design*, 2013, pp. 130–137.
[7] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Design, Automation and Test in Europe*, 2010, pp. 957–960.
[8] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Automation and Test in Europe*, 2013, pp. 1367–1372.
[9] C. Zou, W. Qian, and J. Han, "DPALS: A dynamic programming-based algorithm for two-level approximate logic synthesis," in *International Conference on ASIC*, 2015, pp. 1–4.
[10] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Design Automation Conference*, 2016, pp. 128:1–128:6.
[11] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: systematic logic synthesis of approximate circuits," in *Design Automation Conference*, 2012, pp. 796–801.
[12] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of AIGs for error tolerant applications," in *International Conference on Computer-Aided Design*, 2016, pp. 83:1–83:8.
[13] R. L. Ashenhurst, "The decomposition of switching functions," in *International Symposium on the Theory of Switching*, 1957, pp. 74–116.
[14] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *International Conference on Computer-Aided Desig*, 1997, pp. 78–82.
[15] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 7, pp. 866–876, 2002.
[16] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large Boolean functions via interpolation and satisfiability solving," in *Design Automation Conference*, 2008, pp. 636–641.
[17] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Design Automation Conference*, 2001, pp. 103–108.
[18] B. J. Falkowski and S. Kannurao, "Efficient spectral method for disjoint bi-decompositions of Boolean functions," in *International Symposium on Circuits and Systems*, vol. 2, 2000, pp. 313–316.
[19] T. Sasao, *Switching theory for logic synthesis*. Kluwer Academic Publishers, 1999.
[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, 2001.
[21] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 2, pp. 145–204, 1996.
[22] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.
[23] "Synopsys." [Online]. Available: http://www.synopsys.com/
[24] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification, release 70330." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc