Chapter 7
Approximate Logic Synthesis for FPGA by Decomposition

Zhiyuan Xiang, Niyiqiu Liu, Yue Yao, Fan Yang, Cheng Zhuo, and Weikang Qian

1 Introduction

With the breakdown of Dennard scaling, power consumption has become a bottleneck for circuit design [1]. Meanwhile, many useful applications are inherently error-tolerant. These include machine learning, pattern recognition, and image pro-

This work was supported by the State Key Laboratory of ASIC & System Open Research Grant 2019KF004. Zhiyuan Xiang and Niyiqiu Liu contributed equally.

Z. Xiang · N. Liu
University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China
e-mail: xzy242215@sjtu.edu.cn; lnyq10@sjtu.edu.cn

Y. Yao
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: yueyao@cs.cmu.edu

F. Yang
State Key Laboratory of ASIC & System; Microelectronics Department, Fudan University, Shanghai, China
e-mail: yangfan@fudan.edu.cn

C. Zhuo
College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China
e-mail: czhuo@zju.edu.cn

W. Qian (✉)
University of Michigan-Shanghai Jiao Tong University Joint Institute and MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, Shanghai, China

State Key Laboratory of ASIC & System, Fudan University, Shanghai, China
e-mail: qianwk@sjtu.edu.cn

W. Liu, F. Lombardi (eds.), *Approximate Computing*,
https://doi.org/10.1007/978-3-030-98347-5_7

cessing. Given these trends, approximate computing was proposed as a promising way to design low-power digital circuits for these error-tolerant applications [2]. It relaxes the stringent accuracy requirement to further reduce circuit area, delay, and power consumption. An important area in approximate computing is approximate logic synthesis (ALS), which automatically synthesizes a good approximate circuit under a given error constraint.

Many existing ALS methods are designed for application-specific integrated circuits (ASICs) [3–12], while only few target at field programmable gate arrays (FPGAs) [13, 14]. Modern FPGAs are implemented by a network of lookup tables (LUTs). A LUT of k inputs, known as k-LUT, can implement any k-input Boolean function. This causes fundamental difference between FPGAs and ASICs. The existing ALS methods for ASIC work on a different circuit representation than the LUT network representation. Some ALS methods cannot be applied to handle LUT networks [11], while the others can [5]. Although the former can still be applied in the technology independent synthesis phase for FPGA, they require an additional LUT mapping step to convert the intermediate design into the FPGA design, which leads to a weak control over the final hardware cost. For the latter, they cannot fully exploit the special features of LUT networks. Finally, the few ALS methods for FPGA still rely heavily on the existing LUT mapping tools [13, 14]. Thus, they also have a weak control over the final hardware cost. Therefore, in order to fully explore the power of ALS for FPGAs, it is imperative to develop a method that directly works on the LUT network representation and fully exploits the flexibility of the FPGA designs.

For this purpose, we propose a novel method to perform ALS for FPGAs in this work. Our method aims at reducing the LUT count. It directly works on the LUT network and exploits the reconfigurability of the LUTs to reduce its size.

The basic idea of our method is to approximately implement a LUT subnetwork by the minimum number of LUTs determined by the input size of the subnetwork. To illustrate our idea, consider an optimized LUT subnetwork of 6 inputs shown in the left part of Fig. 7.1. We assume that 3-LUTs are used. This optimal design is implemented by four 3-LUTs. Note that in theory, the minimum number of 3-LUTs needed to implement a 6-input function is 3.[1] However, there does not exist any LUT network of three 3-LUTs to implement the given function, as the LUT network in the left part of Fig. 7.1 is claimed to be optimal. Nevertheless, if we allow errors, it is possible to change the given function properly so that it can be implemented by only three 3-LUTs; one example is shown in the right part of Fig. 7.1. Indeed, there exist several different ways to connect three 3-LUTs. Furthermore, each 3-LUT can implement many different functions. Thus, the number of functions that can be implemented by a network of three 3-LUTs is enormous. On the one hand, this flexibility is helpful, since it is possible for us to find one function very close to the original one, thus with the minimum error introduced. On the other hand, given the large design space, how to efficiently find such a function is a big challenge. Since a

[1] With only two 3-LUTs, we can only realize a function with no more than 5 inputs.

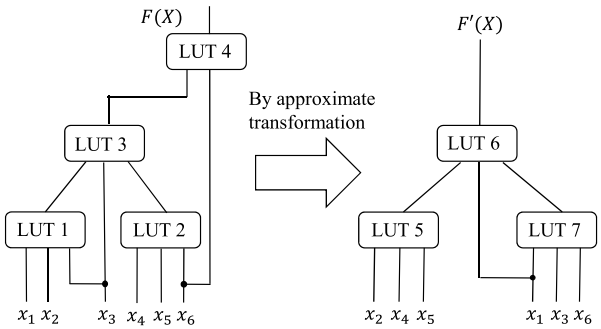

Fig. 7.1 An approximate transformation that reduces the LUT count to the minimum value determined by the input size of the function

LUT network naturally corresponds to a series of decompositions of a function, we propose a decomposition-based technique to solve the above challenge and derive a novel ALS method for FPGA.

Our main contributions are as follows:

- We propose a heuristic method to find an approximate disjoint decomposition for a given function with small error.
- We extend the above method and propose a heuristic method to find an approximate non-disjoint decomposition for a given function with small error.
- We propose an iterative decomposition algorithm that exploits both the approximate disjoint and non-disjoint decompositions to realize a Boolean function with the minimum number of LUTs determined by the input size of the function.
- We design an ALS flow for FPGA based on the iterative decomposition algorithm. Our experimental results showed that the proposed flow achieves more LUT count reduction than the previous state-of-the-art ALS methods for FPGAs.

The rest of the chapter is organized as follows. Section 2 discusses the related works. Section 3 provides the preliminaries. Section 4 presents our proposed methods. Section 5 shows the experimental results. Finally, Sect. 6 concludes the chapter.

2 Related Work

We describe some related works in this section. We first discuss the works related to ALS for FPGA, followed by works related to ALS based on decomposition.

2.1 Approximate Logic Synthesis for FPGA

Wu et al. proposed an ALS method for FPGA designs [13]. The method is based on a heuristic that assumes that by removing some inputs of a function, the final LUT count will drop. Following this heuristic, the method simultaneously removes some inputs of a local subnetwork and modifies its logic function to minimize the error introduced. Once the approximate function is derived, it further applies an existing FPGA mapping tool to the function to obtain the final LUT network. Liu and Zhang also proposed an ALS method and applied it to FPGA synthesis [14]. Their method works on the gate network first and then maps the network into the FPGA design through an FPGA mapping tool. Due to the use of the additional FPGA mapping tool, these previous methods have a weak control over the final hardware cost. In contrast, our method works on the LUT network representation directly to reduce the LUT count.

2.2 Approximate Logic Synthesis Based on Decomposition

Another direction of ALS is to perform approximate decomposition to some local circuits to simplify them. Hashemi et al. [10] proposed a method based on *Boolean matrix factorization*. It approximately factors the truth table of a multi-output function as the product of two Boolean matrices. It then synthesizes an approximate design based on the factorization. Yao et al. [15] proposed a method based on Boolean decomposition. It performs *approximate disjoint bi-decomposition* that recursively separates the input set of a function into two disjoint subsets. However, the method is based on the special disjoint bi-decomposition and thus can only generate networks of 2-input gates. In contrast, our method is based on a general disjoint decomposition and therefore can be applied to synthesize networks of k-LUTs with $k \geq 2$. Furthermore, beyond approximate disjoint decomposition, we also propose an approximate non-disjoint decomposition. Another related work is the proposal of a two-LUT architecture that approximately realizes a given function originally implemented by a single larger LUT [16]. The work also exploits Boolean decomposition to configure the LUTs. However, it targets at LUT-based computation, where a single large LUT implements the target function, which is different from LUT-based FPGA, where many small LUTs are interconnected to realize the target. Due to the architecture difference, the work only requires to perform the approximate disjoint decomposition once. In contrast, our work needs to perform multiple approximate disjoint decompositions recursively.

3 Preliminaries

In this section, we introduce the related preliminaries, including simple disjoint decomposition, fanout-free cone, error measurement, and Monte Carlo simulation.

3.1 Simple Disjoint Decomposition

Our proposed method is based on simple disjoint decomposition, which was pioneered by Ashenhurst [17] and Curtis [18]. We first introduce some definitions.

Definition 1 Let f be a logic function of n variables and $X = \{x_1, \ldots, x_n\}$ be its inputs. Let $\{A, B\}$ be a partition on X. The function f has a *simple disjoint decomposition* with *bound set* A and *free set* B if there exist functions ϕ and F such that $f(X) = F(\phi(A), B)$. The functions F and ϕ are called the *free-set function* and the *bound-set function*, respectively. If the function f has a simple disjoint decomposition, the function is said to be *decomposable*.

Not every logic function is decomposable. Ashenhurst gives a necessary and sufficient condition for the existence of a simple disjoint decomposition under a given partition on the input variables [17]. It is based on a *2-dimensional (2D) truth table* representation of the Boolean function, in which some variables define the columns and the remaining define the rows. An example of the 2D truth table is shown in Fig. 7.2. In what follows, we will also call this representation a *Boolean matrix*. The following theorem gives the necessary and sufficient condition.

Theorem 1 *Let $\{A, B\}$ be a partition on X. A logic function f is decomposable with bound set A and free set B if and only if the Boolean matrix with the variables in A and B defining the columns and the rows, respectively, has at most four distinct types of rows:*

1. *A pattern of all 0s*
2. *A pattern of all 1s*
3. *A fixed pattern p of 0's and 1's*
4. *The complement of the pattern p*

A proof to the above theorem can be found in [19]. We use the following example to illustrate how to obtain the simple disjoint decomposition once the condition in Theorem 1 is satisfied.

Example 1 Figure 7.2 shows a Boolean matrix of a Boolean function $f(x_1, x_2, x_3, x_4)$ with variables x_1 and x_2 defining the rows and variables x_3 and x_4 defining the columns. It satisfies the condition described in Theorem 1: row 1 falls into Type 3, rows 2 and 4 fall into Type 4, and row 3 falls into Type 2. Thus, function f is decomposable with free set as $\{x_1, x_2\}$ and bound set as $\{x_3, x_4\}$. We can set the truth table of the function $\phi(x_3, x_4)$ as the pattern in Type 3. For this example, the truth table is "0110," and correspondingly, $\phi(x_3, x_4) = \overline{x_3}x_4 + x_3\overline{x_4}$. Now, the first,

x_1x_2 \ x_3x_4	00	01	10	11	
00	0	1	1	0	$\phi \wedge \overline{x_1} \wedge \overline{x_2}$
01	1	0	0	1	$\overline{\phi} \wedge \overline{x_1} \wedge x_2$
10	1	1	1	1	$x_1 \wedge \overline{x_2}$
11	1	0	0	1	$\overline{\phi} \wedge x_1 \wedge x_2$

Fig. 7.2 A 2D truth table, or Boolean matrix, of a function f

second, third, and fourth rows of the Boolean matrix represent the functions $\phi\overline{x_1}\,\overline{x_2}$, $\overline{\phi}\overline{x_1}x_2$, $x_1\overline{x_2}$, and $\overline{\phi}x_1x_2$, respectively. Therefore, we obtain the final expression of f as

$$f = \phi\overline{x_1}\,\overline{x_2} + \overline{\phi}\overline{x_1}x_2 + x_1\overline{x_2} + \overline{\phi}x_1x_2 = F(\phi, x_1, x_2).\square$$

3.2 Fanout-Free Cone (FFC)

Our proposed method is based on simple disjoint decomposition. However, it is restricted to single-output Boolean functions. Given this restriction, we apply our method to a particular structure in a circuit called *fanout-free cone (FFC)*. We give the relevant definitions in this section.

We focus on combinational circuits implemented by FGPA. They can be viewed as a directed acyclic graph $N = (V, E)$, where V is the vertex set that contains all LUTs in the circuit and E is the edge set that represents the wire connection among all the LUTs. Given a node $v \in V$ in a graph $N(V, E)$, a *cone* of node v, denoted as C_v, is a subgraph of N consisting of node v and some of its predecessors such that any path from a node in C_v to v lies entirely in C_v [20]. The node v is called the root of C_v. A *fanout-free cone (FFC)* is a cone in which the fanouts of every node other than the root are in the cone [21].

3.3 Error Rate and Monte Carlo Simulation

There are two typical quantities to evaluate the error of an approximate circuit, *error rate (ER)* and *error magnitude (EM)*. ER is defined as the probability of an input pattern that gives an erroneous output for the approximate circuit. EM measures

how much the output of the approximate circuit deviates from the correct output. It is typically used for arithmetic circuits. In this work, we focus on ER.

The number of input combinations of a circuit is exponential to the input size, and thus, it is impractical to enumerate them to calculate the exact ER. Instead, we perform Monte Carlo simulation to obtain an estimation of the ER of an approximate circuit, as is done in many other works [10, 11]. In our implementation, we chose the sample size as 10^5. Besides that, as we will show later, in order to determine proper approximate transformations for a sub-circuit, we need the occurrence probability for each combination of some internal signals. It is also calculated through Monte Carlo simulation. In this case, the simulation results for internal nodes are needed. To speed up our program, we store the simulation results for each node in the memory.

4 Methodology

In this section, we present the proposed method. We begin with an overview of the basic idea, followed by the technical details.

4.1 Basic Idea

Our method works on the FFCs in the given LUT network. An FFC implements a single-output function through a LUT subnetwork. In order to minimize the total LUT count, we minimize the number of LUTs needed to implement a selected FFC. Our method is based on the following observation: the minimum number of k-LUTs needed to implement an n-input function is $\lceil \frac{n-1}{k-1} \rceil$. For example, in order to implement a 7-input (respectively, 8-input) function with 4-LUTs, the minimum number of LUTs needed is 2 (respectively, 3). Although there exists flexibility in the LUT connection and configuration, it may be impossible to exactly realize the given FFC function with the minimum number of LUTs. However, if we introduce minor modification to the original function f, it is possible.

For this purpose, we develop a method to perform approximate disjoint decomposition for a given function. For an arbitrary function, it may not be decomposable. Approximate disjoint decomposition essentially finds a decomposable function close to it. In the context of FPGA synthesis, we repeatedly apply the approximate disjoint decomposition to the function implemented by an FFC. In each round, we derive an approximate disjoint decomposition with a bound set of size k. Assume the obtained bound-set function is ϕ. Then, we implement it by a k-LUT. With this, k inputs in the bound set are replaced by the single signal ϕ. Therefore, the input size of the function is reduced by $(k - 1)$. We repeat this process until the input size of the function is no more than k. At this moment, the final function can be implemented by a single k-LUT. However, for some cases, the number of inputs

of the final function is fewer than k, making the last LUT not fully exploited. To make full use of the last LUT, we propose to connect some signals in the previous levels of the LUT network to the unused inputs of the last LUT. This requires to replace the final approximate disjoint decomposition by an *approximate non-disjoint decomposition*. We also develop a method for this.

Example 2 Consider a LUT network in the left part of Fig. 7.1. It implements a function f with 6 inputs, i.e., $X = \{x_1, \cdots, x_6\}$. If we want to implement f using the minimum number of 3-LUTs, we need two rounds of approximate disjoint decomposition. The intermediate steps are shown in Fig. 7.3a and b. In the first round, suppose that the approximate disjoint decomposition produces an approximation to f as $F_1(\phi_1(A_1), B_1)$ with the bound set $A_1 = \{x_2, x_4, x_5\}$ and the free set $B_1 = \{x_1, x_3, x_6\}$. The corresponding circuit is shown in Fig. 7.3a. After the first round, we find that the function F_1 has 4 inputs ϕ_1, x_1, x_3, x_6. Thus, it cannot be realized by one 3-LUT. Therefore, we continue to the second round of approximate disjoint decomposition. Now $X = \{\phi_1, x_1, x_3, x_6\}$. Suppose that the approximate disjoint decomposition produces an approximation to F_1 as $F_2(\phi_2(A_2), B_2)$ with the bound set $A_2 = \{x_1, x_3, x_6\}$ and the free set $B_2 = \{\phi_1\}$. The resulting circuit is shown in Fig. 7.3b. Now the function F_2 has only two inputs ϕ_1 and ϕ_2. Thus, it can be realized by a 3-LUT. However, the last LUT has an unused input. In this case, we perform an approximate non-disjoint decomposition to change the disjoint decomposition $F_2(\phi_2(x_1, x_3, x_6), \phi_1)$ into a non-disjoint one as $F_3(\phi_3(x_1, x_3, x_6), \phi_1, x_1)$. The final LUT network is shown in Fig. 7.3c. For this example, we reduce both the LUT count and the LUT network depth by one compared to the original LUT network shown in the left part of Fig. 7.1. □

We call the above iterative procedure *iterative approximate decomposition*. It requires $\left(\lceil \frac{n-1}{k-1} \rceil - 1\right) = \lceil \frac{n-k}{k-1} \rceil$ iterations. The number of k-LUTs in the resulting circuit is $\lceil \frac{n-1}{k-1} \rceil$, the minimum achievable value. However, some FFCs in the circuit may already contain the minimum number of LUTs. In this case, to further reduce their LUT counts, we need to remove some of their inputs. For this purpose, we also introduce an iterative input removal method. It can be treated as a special case of the iterative approximate decomposition.

In the following, we will describe details of the approximate disjoint and non-disjoint decompositions in Sects. 4.2 and 4.3, respectively. Then, we will describe the iterative approximate decomposition in Sect. 4.4, followed by the iterative input removal in Sect. 4.5. Finally, we will show the overall ALS flow in Sect. 4.6.

4.2 Approximate Disjoint Decomposition

In this section, we present the approximate disjoint decomposition for a fixed bound set and free set. We note that a similar method is presented in [16]. The problem solved by this technique is formally stated as follows: *given a Boolean function*

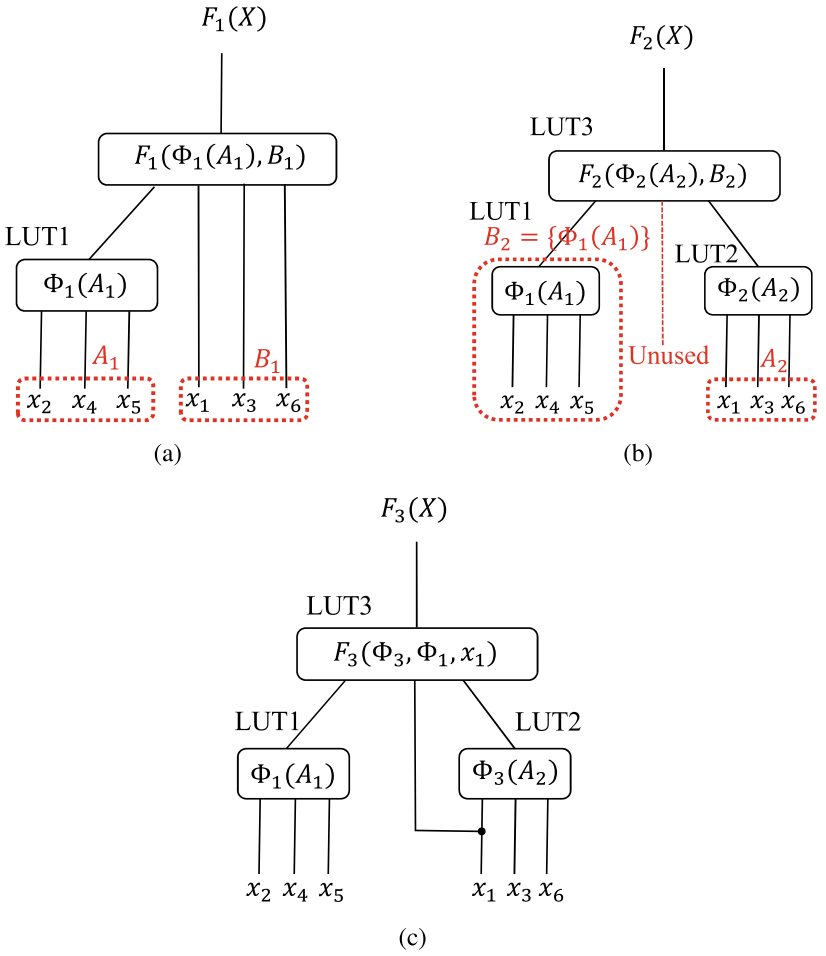

Fig. 7.3 An illustration of the proposed iterative approximate decomposition. (**a**) Applying the first round of approximate disjoint decomposition. (**b**) Applying the second round of approximate disjoint decomposition. (**c**) Applying an approximate non-disjoint decomposition

$f(X)$ and a partition (X_1, X_2) of the input set X, find a decomposable function $F(\phi(X_1), X_2)$ with the smallest ER over f. In what follows, if the bound set and the free set are clear from the context, we will also represent the decomposable function by a pair (F, ϕ) for simplicity.

The solution works on the Boolean matrix of the given function f with the variables in X_1 and X_2 defining the columns and rows, respectively. Besides that, in order to evaluate the ER of the approximate function over the original function, we also need to know the occurrence probability of each input pattern of the function f. For this purpose, we augment the Boolean matrix by including

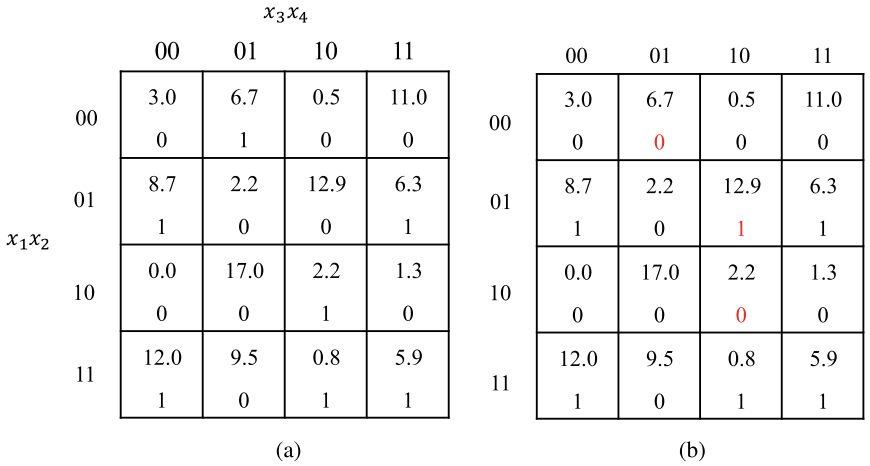

Fig. 7.4 Augmented matrices. (**a**) A non-decomposable function. (**b**) A decomposable function that approximates the function in (**a**)

the occurrence probability of each input pattern. We call the resulting matrix an *augmented matrix*. An example of this is shown in Fig. 7.4a. In each entry of the matrix, the binary value represents the output of the function and the real value represents the occurrence probability of each input pattern. For simplicity, we actually show the occurrence percentage in the matrix. The occurrence probability of each input pattern is obtained through the Monte Carlo simulation described in Sect. 3.3. For example, consider a function on local inputs x_1, x_2, and x_3. Assume that the Monte Carlo simulation has M samples. Then, the occurrence probability for $(x_1, x_2, x_3) = (a_1, a_2, a_3)$, where $a_1, a_2, a_3 \in \{0, 1\}$, is calculated as $C(a_1, a_2, a_3)/M$, where $C(a_1, a_2, a_3)$ is the number of times in the Monte Carlo simulation when $(x_1, x_2, x_3) = (a_1, a_2, a_3)$.

Given an arbitrary function, a fixed bound set, and a fixed free set, the function may not be decomposable with the bound set and the free set. However, if we make some proper changes to the entries in the Boolean matrix, we can construct a decomposable function by introducing some error. The ER is calculated as

$$\varepsilon = \sum_i \sum_j |B'[i][j] - B[i][j]| \cdot P[i][j], \tag{7.1}$$

where $B[i][j]$'s and $B'[i][j]$'s are the Boolean entries in the augmented matrices of the original function and the decomposable function, respectively, and $P[i][j]$'s are the probability entries in the augmented matrix of the original function. In other words, the ER is the sum of the occurrence probabilities of those input patterns with an output change.

Example 3 Given the function, the bound set, and the free set shown in Fig. 7.4a, the function is not decomposable with the bound set and the free set according to Theorem 1. However, by flipping the outputs of some input patterns, we can construct a decomposable function. A possible example is shown in Fig. 7.4b. The changed bits are labeled in red. By Eq. (7.1), the ER of the approximation is $\varepsilon = (6.7 + 12 + 2.2)\% = 20.9\%$. □

By Theorem 1, for a fixed bound set A and free set B, the Boolean matrix of a decomposable function is fully determined by two factors. The first is the fixed pattern p in Theorem 1, which we call *a pattern vector*. Note that $p \in \{0, 1\}^{2^{|A|}}$. As shown in Example 1, the pattern vector determines the bound-set function. The second is the collection of the type indices of all the rows. We represent it as a vector $r \in \{1, 2, 3, 4\}^{2^{|B|}}$, which we call *a row-type vector*. The ith entry in the row-type vector represents one of the four types that the ith row belongs to. For example, for the decomposable function shown in Fig. 7.4b, its pattern vector is $p = (1, 0, 1, 1)$ and its row-type vector is $r = (1, 3, 1, 3)$.

In order to determine the decomposable function for a fixed bound set and free set with the smallest ER over the original function, it is equivalent to finding the optimal pair of pattern vector and row-type vector (p, r). However, there are $2^{2^{|A|}} \cdot 4^{2^{|B|}}$ possible pairs in the solution space. A brute-force enumeration is prohibitive. Instead, we propose an algorithm to search for a good local optimal solution. The algorithm is based on the following two observations.

1. Once the pattern vector p is fixed, we can identify an optimal row-type vector r efficiently. To do this, we only need to decide each entry in the optimal row-type vector. To determine the ith entry, we compare the ith row of the original Boolean matrix with four choices, which are a pattern of all 0s, a pattern of all 1s, pattern p, and the complement of pattern p, and obtain the ER for each choice. The final best choice for the ith entry is just the one with the smallest ER.
2. Once the row-type vector r is fixed, we can identify an optimal pattern vector p efficiently. To do this, we only need to decide each entry in the optimal pattern vector. Consider the ith entry in the pattern vector. It has only two choices, 0 and 1. We enumerate these two choices. For each choice, since the row-type vector is fixed, we can obtain the ith column of the Boolean matrix B' of a decomposable function determined by the row-type vector r and the pattern vector p with the ith entry as that choice. We then obtain the ER of the ith column of B' over that of the original Boolean matrix. We compare the ERs for the two choices and select the choice giving a smaller ER.

Example 4 Consider the Boolean matrix shown in Fig. 7.4a. Suppose the fixed pattern vector $p = (1, 0, 1, 1)$. Now, we decide the entries in the optimal row-type vector r one by one. With $r[1]$ chosen as $1, 2, 3, 4$, the first row in the new Boolean matrix is $(0, 0, 0, 0)$, $(1, 1, 1, 1)$, $(1, 0, 1, 1)$, and $(0, 1, 0, 0)$, respectively. Comparing these four choices with the first row in the original Boolean matrix, we can obtain their ERs as 6.7%, 14.5%, 21.2%, and 0, respectively. Since the

Algorithm 1: Function *ApxDecomp* for finding a decomposable function with a given bound set and free set that has a small ER over the given Boolean function

Input : An augmented matrix M specifying the given Boolean function, the bound set, the free set, and the occurrence probability of each input pattern, and a parameter T.
Output : A decomposable function (F, ϕ).

```
1  IniPSet ⇐ ∅;
2  for each row p in M.B do
3  |  calculate the optimal row-type vector r with the pattern vector set as p;
4  |  p.error ⇐ ER of the decomposable function determined by p and r;
5  choose T distinct rows in M.B with the smallest ERs and add them into IniPSet;
6  OptimSet ⇐ ∅;
7  for each pattern vector p in IniPSet do
8  |  while p and r have been updated do
9  |  |  fix p and calculate the optimal r;
10 |  |  fix r and calculate the optimal p;
11 |  add the decomposable function (F, φ) determined by p and r into OptimSet;
12 return the decomposable function (F, φ) in OptimSet with the minimum ER;
```

last choice has the minimum ER, we choose $r[1]$ as 4. The other entries of r are determined similarly. The final optimal row-type vector is $r = (4, 3, 1, 3)$.

Now, suppose the fixed row-type vector $r = (4, 3, 1, 3)$. We decide the entries in the optimal pattern vector p one by one. We use $p[3]$ as an example. For $p[3] = 0, 1$, the third column in the new Boolean matrix is $(1, 0, 0, 0)^T$ and $(0, 1, 0, 1)^T$, respectively. Comparing these two choices with the third column in the original Boolean matrix, we can obtain their ERs as 3.5% and 15.1%, respectively. Thus, we choose $p[3] = 0$. The other entries are determined similarly. The final optimal pattern vector is $p = (1, 0, 0, 1)$. □

The above two observations lead to a method to search for a local optimal solution. Instead of searching for p and r simultaneously, we first fix p and optimize r. Then, we fix r and optimize p. In this way, we keep updating p and r until they do not change. Then, we reach a local optimal solution.

Based on the above idea, we propose an algorithm for finding a decomposable function with a given bound set and free set that has a small ER over the original function f. It is shown in Algorithm 1. The function takes an augmented matrix M as inputs. The matrix specifies the given Boolean function, the bound set, the free set, and the occurrence probability of each input pattern. The algorithm has two major parts. The first part creates multiple initial pattern vectors p's (see Lines 2–5). This is important because there may exist many local minima in which our basic optimization algorithm may get stuck. In order to improve the quality, one way is to choose multiple initial starting points. In our implementation, we choose the initial pattern vectors p's from the existing rows. For this purpose, we visit each row p in the Boolean matrix $M.B$ of the augmented matrix, obtain the associated optimal row-type vector r, and calculate the ER for this pair of p and r. We choose T distinct row patterns p's that give the smallest ERs as the initial pattern vectors p's.

The second part of the algorithm visits each initial p selected from the first part. For each p, it performs optimization on p and r alternatively until a local minimum is reached (see Lines 8–10). Then, it stores the corresponding decomposable function (F, ϕ) into the set *OptimSet* (see Line 11). Finally, it returns the decomposable function (F, ϕ) with the smallest ER in the set *OptimSet*.

4.3 Approximate Non-disjoint Decomposition

The approximate disjoint decomposition has the restriction that the bound and free sets are disjoint. Depending on the input size of the given FFC, some LUTs in the final LUT netlist obtained by the approximate disjoint decomposition may have some unused inputs. For instance, as shown in Example 2, if only using approximate disjoint decomposition, the last LUT *LUT3* has one input unused. *LUT3* together with *LUT2* implements the last approximate disjoint decomposition of the form $F_2(\phi_2(x_1, x_3, x_6), \phi_1)$. If we choose an input from *LUT2* and connect it to the unused input, as shown in Fig. 7.3c, the circuit area and depth do not change. By properly configuring the functions of these two LUTs, we can possibly reduce the ER. However, the resulting decomposition is not a disjoint decomposition anymore. It is a non-disjoint decomposition of the form $F(\phi(A'), B')$, where $A' \bigcap B' \neq \emptyset$. The number of possible non-disjoint decompositions is even larger than the number of disjoint ones. The search for an optimal one is time-consuming. We propose a fast solution for a good approximate non-disjoint decomposition based on an existing approximate disjoint decomposition.

Assume that an existing approximate disjoint decomposition has its bound set as A and free set as B. The original function has a corresponding augmented matrix with the variables in the sets A and B defining the columns and rows, respectively. To create a non-disjoint decomposition, a variable from the bound set A will be added to the free set B, creating a larger free set B'. Correspondingly, we first update the original augmented matrix. We call the resulting matrix the *updated augmented matrix*. We use the following example to illustrate how to obtain the updated augmented matrix.

Example 5 Suppose that the target function is $f(x_1, x_2, x_3)$ and the bound set A and the free set B of an existing approximate disjoint decomposition are $A = \{x_1, x_2\}$ and $B = \{x_3\}$. Figure 7.5a shows the augmented matrix of f with the variables in A defining the columns and the one in B defining the rows. We assume that the input combination (x_1, x_2, x_3) follows a uniform distribution. Thus, the occurrence probability of each input pattern is $1/8 = 12.5\%$.

Without loss of generality, suppose that we add x_2 into the free set $\{x_3\}$ to construct an approximate non-disjoint decomposition. We also let it be the first variable in the free set. With x_2 added into the free set, we modify the augmented matrix. The updated one is shown in Fig. 7.5c. The height of the augmented matrix is doubled. The upper half has $x_2 = 0$, while the lower half has $x_2 = 1$. The Boolean entries

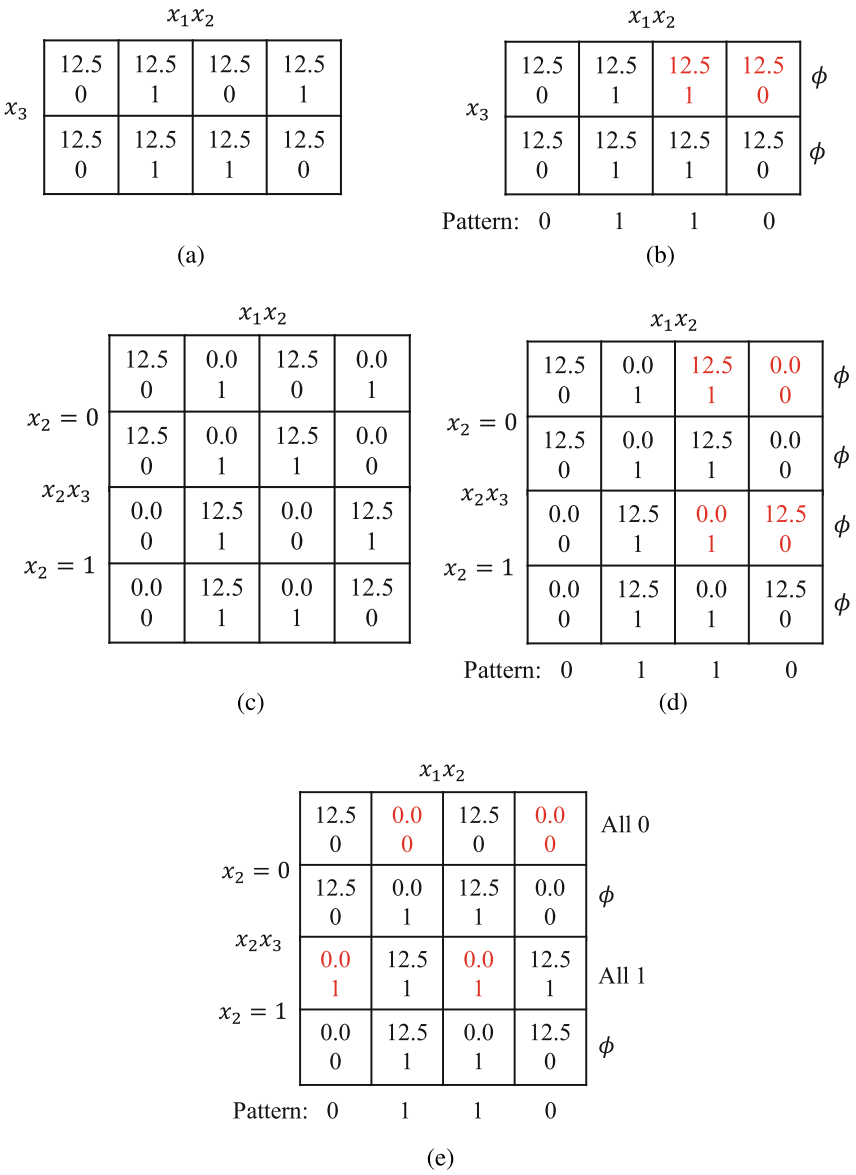

Fig. 7.5 An illustration of obtaining an approximate non-disjoint decomposition from an existing approximate disjoint decomposition. The approximate outputs different from the original ones are highlighted in red. (**a**) The augmented matrix for a function f. (**b**) The augmented matrix for the closest approximate disjoint decomposition to f. (**c**) The updated augmented matrix for the one in (**a**) with x_2 added into the free set. (**d**) The updated augmented matrix for the one in (**b**) with x_2 added into the free set. (**e**) The augmented matrix for the closest approximate non-disjoint decomposition to f

of the upper and lower half of the new matrix are the same as those of the original augmented matrix. However, the probability entries are changed. For the entries where the x_2's in the row and the column take different values, their probability values are 0, since these combinations can never occur, while for the entries where x_2's in the row and the column take the same value, their probability values are $1/8$. □

For the existing approximate disjoint decomposition of the original function, it also has a corresponding augmented matrix M_F. Suppose that the Boolean part of the augmented matrix is characterized by a pattern vector p and a row-type vector r. With a variable x from the bound set added into the free set, we can also obtain an update augmented matrix from M_F using the same method described above. The Boolean part of the updated augmented matrix is characterized by a pattern vector p' and a row-type vector r'. Assume that x is added as the first variable in the free set. Then, it can be easily seen that $p' = p$ and r' is two r's cascaded together.

Example 6 For the function shown in Fig. 7.5a, Fig. 7.5b shows the augmented matrix M_F of an approximate disjoint decomposition with the lowest ER. The Boolean part of the matrix is characterized by the pattern vector $p = (0, 1, 1, 0)$ and the row-type vector $r = (3, 3)$.The ER of this decomposition is $1/4$.

With x_2 added into the free set as its first variable, we can also obtain the updated augmented matrix as shown in Fig. 7.5d from M_F. The Boolean part of the updated augmented matrix is characterized by the pattern vector $p' = (0, 1, 1, 0)$ and the row-type vector $r' = (3, 3, 3, 3)$. Clearly, $p' = p$ and r' is two r's cascaded together. □

To derive an approximate non-disjoint decomposition, we work on the updated augmented matrix of the original function. Similar as the disjoint case, a non-disjoint decomposition can also be characterized by a pattern vector and a row-type vector. Thus, we only need to find an optimal pattern vector p^* and row-type vector r^*. By the above discussion, we can see that the existing approximate disjoint decomposition gives an initial solution for the non-disjoint decomposition with the pattern vector as p' and the row-type vector as r'. Then, by applying the alternative pattern and row-type vectors updating mechanism described in Sect. 4.2, we are guaranteed to find a non-disjoint decomposition with ER no more than that of the given approximate disjoint decomposition.

Example 7 For the function shown in Fig. 7.5a, the existing approximate disjoint decomposition gives an initial solution for the non-disjoint decomposition as shown in Fig. 7.5(d). By Example 6, the initial pattern vector is $p' = (0, 1, 1, 0)$ and the initial row-type vector is $r' = (3, 3, 3, 3)$. By applying the alternative pattern and row-type vectors updating mechanism, we can eventually derive a non-disjoint decomposition shown in Fig. 7.5e. Its pattern vector is $p = (0, 1, 1, 0)$ and its row-type vector is $r = (1, 3, 2, 3)$. From the pattern and row-type vectors, we can get the final approximate non-disjoint decomposition as $F(\phi, x_2, x_3) = x_2\overline{x_3} + x_3\phi$ with $\phi = \overline{x_1}x_2 + x_1\overline{x_2}$. Its ER is 0. □

Note that the above discussion assumes that only one variable from the bound set is added into the free set. However, the method can also be extended to add more than one variable from the bound set into the free set. Also, the discussion is on a special case where the input added into the free set is a direct fanin variable x_i of the bound-set function $\phi(x_1, \ldots, x_n)$. However, the method can also be extended when the input is a transitive fanin of the bound-set function. For this general case, the probability entries in the updated augmented matrix should be set as the occurrence probabilities of the corresponding input patterns, which can be obtained by the Monte Carlo simulation.

4.4 Iterative Approximate Decomposition

In this section, we present the details on how we obtain a structure with the fewest LUTs to implement a given function corresponding to an FFC in the circuit. It exploits both the approximate disjoint and non-disjoint decomposition techniques described above. It first builds a LUT network with the fewest LUT only using the approximate disjoint decomposition. However, the last LUT may have some unused inputs. Then, it applies the approximate non-disjoint decomposition to exploit the unused inputs to further reduce ER.

The proposed approximate disjoint decomposition method works under the assumption that the bound set and the free set are given. Before we apply it to find a good approximate decomposition, we need to decide the bound set. Suppose that the given function has n inputs. By the basic idea described in Sect. 4.1, we need to select a bound set of size k. Therefore, there are $\binom{n}{k}$ choices for the bound set in total. Different bound sets lead to different decomposable functions $F(\phi(X_1), X_2)$ and the associated ERs.

Furthermore, as we stated in Sect. 4.1, for a typical function, we need to do multiple rounds of approximate disjoint decomposition. This brings another problem. That is, the different choices made at the previous rounds influence the later choices. For example, in Example 2, in the first round, we choose bound set $A_1 = \{x_2, x_4, x_5\}$, and by applying the approximate disjoint decomposition, we obtain the decomposable function $F_1(\phi_1(A_1), B_1)$. Then, in the second round, the target function for the decomposition is F_1. In contrast, if we choose the bound set $A_1' = \{x_1, x_3, x_5\}$ in the first round, then by applying the approximate disjoint decomposition, we obtain another decomposable function $F_1'(\phi_1'(A_1'), B_1')$. Then, in the second round, the target function for the decomposition is F_1'. This will lead to a different final solution.

In order to address the above issues, we propose an iterative approximate decomposition algorithm based on local beam search [22]. The idea is that in each round of the decomposition loop, we always keep $m \geq 1$ promising decomposable functions

$$F_1(\phi_1(A_1), B_1), \ldots, F_m(\phi_m(A_m), B_m).$$

Algorithm 2: The proposed iterative approximate decomposition

Input: Simulation result *sim*, a given FFC C, and parameters T and m.
Output: An approximate LUT network with the minimum LUT count.

```
1  TopChoice ⇐ {C};
2  TopChoice[1].H ⇐ the Boolean function of C;
3                  ▷ TopChoice[1] is the first element in the set TopChoice;
4  n ⇐ the input size of C;
5  for i = 1 to ⌈(n − k)/(k − 1)⌉ do
6      MaxHeap ⇐ ∅;
7                  ▷ MaxHeap stores the approximate circuits. Its key is the ER;
8      for j = 1 to |TopChoice| do
9          for each partition (bSet, fSet) on the input set of TopChoice[j].H do
10             M ⇐ createMatrix(sim, bSet, fSet, TopChoice[j]);
11             (F, φ) ⇐ ApxDecomp(M, T);
12             NewCkt ⇐ apply(TopChoice[j], F, φ, bSet, fSet);
13             NewCkt.calculateError(sim);
14             MaxHeap.insert(NewCkt);
15             if |MaxHeap| > m then
16                 MaxHeap.deleteMax();
17     TopChoice ⇐ ∅;
18     for j = 1 to |MaxHeap| do
19         insert the circuit corresponding to the free-set function of MaxHeap[j] into
           TopChoice;
20 if (n − 1) is not a multiple of (k − 1) then
21     for j = 1 to |TopChoice| do
22         TopChoice[j].ApxNonDisjuncDecomp();
23 return the approximate circuit in TopChoice with the smallest ER;
```

In the next round, we obtain all approximate decomposable functions derived from $F_1, \ldots, F_m$ and keep the top m based on the smallest ERs.

The algorithm is shown in Algorithm 2. It takes an FFC as its input. Suppose the input size of the FFC is n. As we stated in Sect. 4.1, the iterative approximate decomposition needs $\lceil (n-k)/(k-1) \rceil$ rounds. In each round, there is a set of partial LUT networks obtained from the previous round. They are stored in the set *TopChoice*. Each element in *TopChoice* also has a data member H storing the latest function to be decomposed. We iterate over all elements in *TopChoice* (see Line 8). For the jth element *TopChoice*$[j]$, we further iterate over all pairs of bound set and free set partitioned from the input set of the function H of *TopChoice*$[j]$ (see Line 9). For each pair of bound set *bSet* and free set *fSet*, the function *createMatrix* prepares the augmented matrix M from the bound set, the free set, the simulation trace, and the current partial LUT network *TopChoice*$[j]$ (see Line 10). Then, the function *ApxDecomp* shown in Algorithm 1 is called to generate a good decomposable function that approximates the function H associated with *TopChoice*$[j]$ (see Line 11). After that, the obtained decomposable function is applied to the partial circuit *TopChoice*$[j]$ to derive a new circuit *NewCkt* (see Line 12). Then, the ER of the new circuit is calculated (see Line 13) before it is inserted into a max heap *MaxHeap* (see Line 14). The max heap is indexed on the

ER of the circuit. If its size is larger than m, then the element in it with the largest ER is removed (see Lines 15–16). This essentially ensures that in each round, at most m candidates with the smallest ERs are kept. After all the elements in the set *TopChoice* have been visited, the set *TopChoice* is first reset to an empty set (see Line 17), and then, the circuit corresponding to the free-set function of each element in *MaxHeap* is inserted into *TopChoice* (see Lines 18–19). This leads to the next round.

After the entire decomposition loop finishes, we obtain m LUT networks with the minimum LUT count only through the approximate disjoint decomposition. If $(n - 1)$ is not a multiple of $(k - 1)$, then the last k-LUT of each LUT network in the set *TopChoice* will have some unused inputs. In this case, for each LUT network, we apply the approximate non-disjoint decomposition to further reduce its ER, while keeping the circuit area and depth (see Lines 20–22). The non-disjoint decomposition is based on the disjoint decomposition for the last k-LUT. To reduce the search space, the additional inputs added into the free set are chosen as the primary inputs of the FFC that are different from the inputs in the current free set. We iterate over all possible input choices. For each choice, we build the updated augmented matrix with its probability entries obtained from the Monte Carlo simulation. Then, the solution described in Sect. 4.3 is applied to find an optimal non-disjoint decomposition for that input choice. Once all choices of the primary inputs are traversed, the non-disjoint decomposition with the lowest ER is selected. Finally, the approximate LUT network with the smallest ER in the set *TopChoice* is returned (see Line 23).

4.5 Iterative Input Removal

The proposed iterative approximate decomposition can effectively reduce the LUT count of an FFC if the LUT count of the original FFC is more than the minimum value $\lceil \frac{n-1}{k-1} \rceil$. However, some FFCs in a circuit may already contain the minimum number of LUTs. We call them *optimal-size FFCs*. For these FFCs, if we want to further reduce their LUT counts, we need to reduce the number of inputs. The way we reduce the number of inputs of FFCs is called *input removal*. One basic requirement for input removal is that the ER introduced should be minimal.

We can exploit the proposed approximate disjoint decomposition to remove any given set of inputs and obtain a function on the remaining inputs with the smallest ER over the original function. In the approximate disjoint decomposition, if we only allow each row to be either all 0s or all 1s, then the resulting function is independent of the bound-set inputs. For example, in a disjoint decomposition with the bound set as $\{x_1, x_2\}$ and the free set as $\{x_3, x_4\}$, if the first and fourth rows of the Boolean matrix are all 0 and the second and third rows are all -1, then the function given by the decomposition is $F(x_1, x_2, x_3, x_4) = \overline{x_3}x_4 + x_3\overline{x_4}$, independent of the variables x_1 and x_2.

Thus, if we want to remove a set of inputs, we can set them as the bound set and solve for a special case of the approximate disjoint decomposition where each entry

of the row-type vector is only limited to either 1 or 2. This will give a function without these inputs and with the lowest ER. However, another problem is that we do not know which set of inputs we should remove to obtain the lowest ER. Theoretically, this can be determined by examining all input combinations, but it will lead to an exponential complexity. To improve the efficiency of the algorithm, instead, we do the input removal iteratively. In each iteration, the bound set size is fixed to 1 and the input choice with the lowest ER is obtained and removed. The iteration repeats until we cannot remove an input without letting the error accumulated so far exceed a given threshold ε, which is set as 0.1 times the given ER threshold in our implementation. This process is called *iterative input removal*. In the extreme case where all inputs of the FFC are removed, we essentially reduce the FFC to a constant. In our overall ALS flow, besides the optimal-size FFC, we also apply the iterative input removal to nonoptimal-size FFCs, which can also help reduce the LUT count.

4.6 Overall ALS Flow

In this section, we present the overall ALS flow integrating our proposed techniques. It is shown in Fig. 7.6a. During the process, we keep an ER margin, initialized as the given ER threshold (see Box 1), to help us ensure that the ER of the approximate circuit does not exceed the ER threshold. We perform multiple synthesis rounds. In each round, we first perform global Monte Carlo simulation (see Box 3), which will be used later to obtain the probability entries in the augmented matrices for various local FFCs. Then, we traverse the nodes in a topological order (see Box 4). For each node, we visit all of its candidate FFCs (see Box 5). In our study, we consider FPGA technology using 4-LUTs, that is, $k = 4$. Given this k value, we found that our proposed iterative approximate decomposition is runtime-efficient when the input size is no more than 12. Also, a nontrivial FFC in the LUT network should have at least $(k + 1)$ inputs. Thus, we limit the candidate FFCs to those with the input sizes from 5 to 12.

For each candidate FFC, we obtain a local approximate transformation for it (see Box 5). The details of this step are shown in Fig. 7.6b. It applies the proposed iterative input removal followed by the iterative approximate decomposition. For each node n, after all of its FFCs are visited, we first eliminate those FFCs with the local transformations that would increase the delay of n if the transformation is applied. This guarantees that our method does not increase the depth of the LUT network. We also eliminate those FFCs such that the ERs of their local transformations exceed the ER margin. This guarantees that the approximate transformation to be selected will not let the ER of the circuit exceed the ER threshold. For the remaining FFCs, we choose the FFC c with the largest score and replace c by its local transformation (see Box 6). The score for an FFC is defined as l/e, where l is the number of LUTs we can reduce by replacing the FFC with its local transformation and e is the ER of that transformation. By our scoring

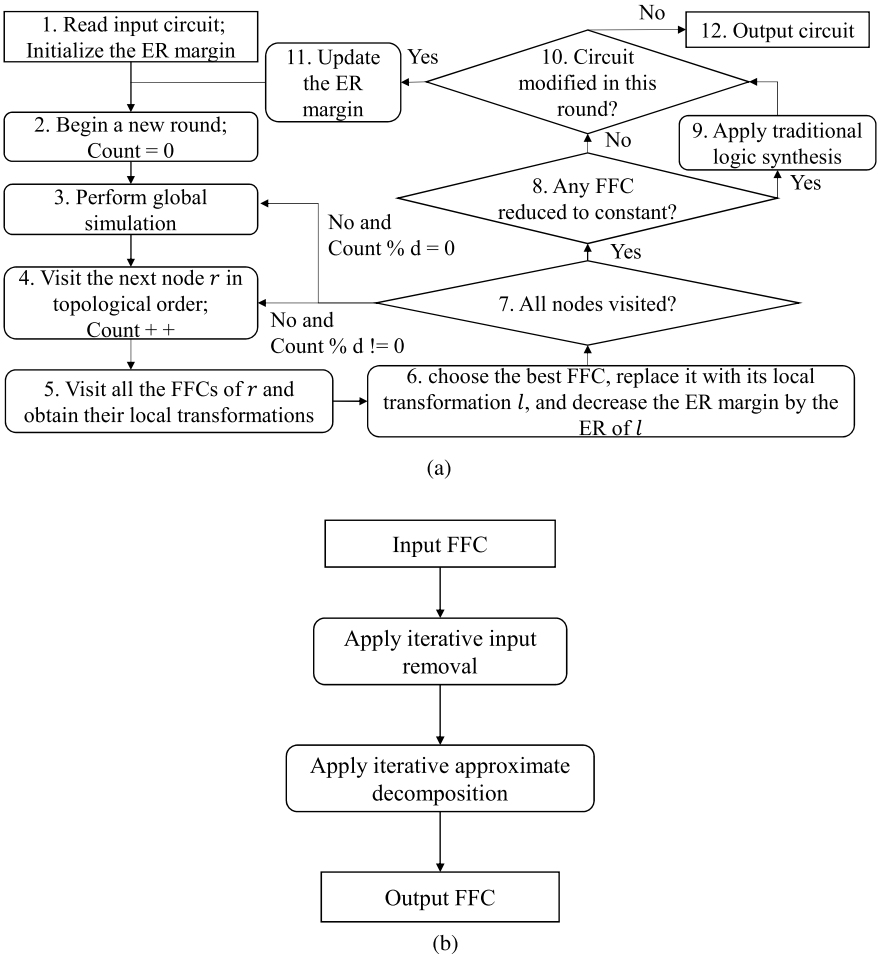

Fig. 7.6 The proposed approximate logic synthesis flow for FPGAs. (**a**) The proposed flow. (**b**) The flow to obtain a local approximate transformation

mechanism, this change maximizes the LUT count reduction per ER introduced. After the change, we decrease the ER margin by the ER of the selected local transformation (see Box 6). Then, we visit the next node in the topological order until all the nodes have been visited.

Within each round, if we re-simulate the whole circuit each time we change an FFC, it would take a long time. However, if we do not, for any FFC that overlaps with the modified one and has not been visited, the occurrence probabilities of its input combinations may not be accurate. To balance the runtime with the accuracy, we count the nodes we have visited in the current round. When d more nodes have been visited, we will perform global Monte Carlo simulation. In our

implementation, we set d as 10. Also, for any FFC that overlaps with an FFC that has been modified after the last global simulation, we will skip it.

After all nodes in the LUT network are visited, we check whether any FFC has been modified to a constant value as a special case of the local approximate transformation (see Box 8). If so, we will apply traditional logic synthesis method to propagate the constant and further simplify the circuit (see Box 9). Then, we check whether in the current round, the circuit has been modified (see Box 10). If not, it means that there is no chance to further improve the circuit and we return the circuit as the final result (see Box 12). Otherwise, we begin a new round and update the ER margin by decreasing it by the actual ER (see Box 11). This update is because the ER of an approximate transformation is measured at the output of an FFC. However, due to the logic masking effect, an error at the output of an FFC may not be observed at the primary outputs of the circuit [11]. Thus, the actual ER could be smaller. Therefore, before the next round starts, we obtain the actual ER through simulation and update the ER margin as the given ER threshold minus the current ER.

5 Experimental Results

In this section, we present the experimental results. We implemented our algorithm in C++ and tested it on a computer with 3.4 GHz CPU and 8GB memory. The parameters T and m in Algorithm 2 were both set as 5.

Our benchmark circuits include all the random control circuits in the EPFL benchmark suite with the LUT count smaller than 1000. Besides them, in order to compare our method with the state-of-the-art ALS methods for FPGA [13, 14], we also included the same MCNC circuits used in [13] and [14]. The original circuits were mapped into networks of 4-LUTs using the logic synthesis tool ABC [23]. We executed the mapping command "if -K 4" multiple times to get the FPGA circuits with the minimum numbers of LUTs as the inputs to our algorithm.

5.1 The Performance of Our Method

Tables 7.1 and 7.2 show the synthesis results of our ALS method for the MCNC and EPFL benchmarks, respectively, under 5% ER constraint. The columns "size" and "depth" list the LUT network size, measured by the number of used LUTs, and the LUT network depth, respectively. The columns under "our baseline" list the sizes and depths of the well-optimized circuits as the inputs to our ALS method. The column "SRR" lists the size reduction ratio (SRR), calculated as the ratio of the LUT number reduction over the LUT number of the input circuit. The arithmetic mean SRRs for the two sets of benchmarks are 24.9% and 29.3%, respectively. For most benchmarks, our method can achieve more than 10% of size reduction.

Table 7.1 The results of our method and the previous methods [13, 14] for the MCNC benchmarks under the ER threshold of 5%

	Our baseline		Our method				Baseline [13, 14]		SCALS [14]			Wu [13]	
Circuit	Size	Depth	Size	Depth	SRR	Runtime (s)	Size	Depth	Size	Depth	SRR	Size	SRR
C432	68	11	61	11	0.103	175.4	97	10	55	10	**0.433**	79	0.186
C880	117	8	97	8	0.171	72	128	8	107	8	0.164	102	**0.203**
C1908	117	9	46	3	**0.607**	32	122	9	88	9	0.279	50	0.590
C2670	210	7	148	7	**0.295**	896.7	295	7	224	7	0.241	252	0.146
C3540	351	12	332	11	0.054	1829.3	346	12	305	11	**0.118**	325	0.061
C5315	465	8	450	8	0.032	2954.7	503	9	439	9	**0.127**	468	0.070
C7552	596	8	410	8	**0.312**	2566.6	593	8	440	8	0.258	486	0.180
Alu4	689	7	421	7	0.389	216	710	7	411	7	**0.421**	483	0.320
Alu2	198	10	150	8	**0.242**	442.3	160	12	135	11	0.156	136	0.150
Apex6	247	6	197	4	0.202	668.4	253	6	210	6	0.170	197	**0.221**
dalu	445	11	297	8	**0.333**	1100.6	425	11	329	11	0.226	349	0.179
Mean	318	8.8	237	7.5	**0.249**	995.8	330	9	249	8.8	0.236	266	0.210

Table 7.2 The result of our methods for the EPFL benchmarks under the ER threshold of 5%

	Our baseline		Our method			
Circuit	Size	Depth	Size	Depth	SRR	Runtime (s)
ALU ctrl	58	3	52	3	0.103	11.4
cavlc	299	6	248	6	0.171	365.5
Decoder	305	2	290	2	0.049	14.7
i2c	376	5	246	4	0.346	36.7
int2float	68	6	59	6	0.132	50.8
Priority	183	42	25	4	0.863	104.8
Router	72	5	31	1	0.569	3.2
Arbitor	650	23	286	13	0.560	36.7
mean	251	11.5	165	8.8	0.293	78.0

The effectiveness of our method is closely related to the number of nonoptimal-size FFCs in the input circuit. For some circuits like `decoder` in Table 7.2, they have few nonoptimal-size FFCs. With an insufficient number of such FFCs fed into the iterative approximate decomposition algorithm, our method cannot find approximate transformations with small ERs to further improve the circuit size. Thus, the size reduction for them is limited.

In our method, we need to re-simulate the whole circuit after applying some approximate transformations to the circuit. This process is repeated until the ER of the approximate circuit exceeds the error threshold. Before each round of re-simulation, we record the size of the current LUT network together with its ER. The plots of SRR versus ER are shown in Figs. 7.7 and 7.8 for the MCNC and the EPFL benchmarks, respectively. For all the circuits, the size reduction gradually increases as the ER increases to 5%. For our method, if we introduce $x\%$ error into the circuit, we can typically gain a $2.5\times\%$ to $4\times\%$ size reduction. For some circuits such as `priority` and `router`, the LUT network sizes reduce significantly with a small amount of error introduced. Furthermore, for some circuits (e.g., `priority` and `router`), the approximation process stops before the ER threshold 5% is reached because at these stopping points, our algorithm cannot find nonoptimal-size FFCs in the circuits or the remaining few nonoptimal-size FFCs only have local transformations with ERs larger than the remaining ER margin.

Tables 7.1 and 7.2 also list the running time of our method. As the running time depends not only on the size of the circuit but also on the number of iterations in the synthesis procedure, we only show the average running time of one iteration for each circuit. The average running time theoretically is proportional to the number of FFCs in the circuit, which is roughly proportional to the number of LUTs of the circuit. However, there are some exceptions, as they have too many or too few FFCs compared to their LUT sizes (e.g., `decoder`). Also, some circuits may only have small-size FFCs, which lead to extremely short running time compared to LUT sizes of the circuits, like `Alu4`, `i2c`, `router`, and `arbitor`.

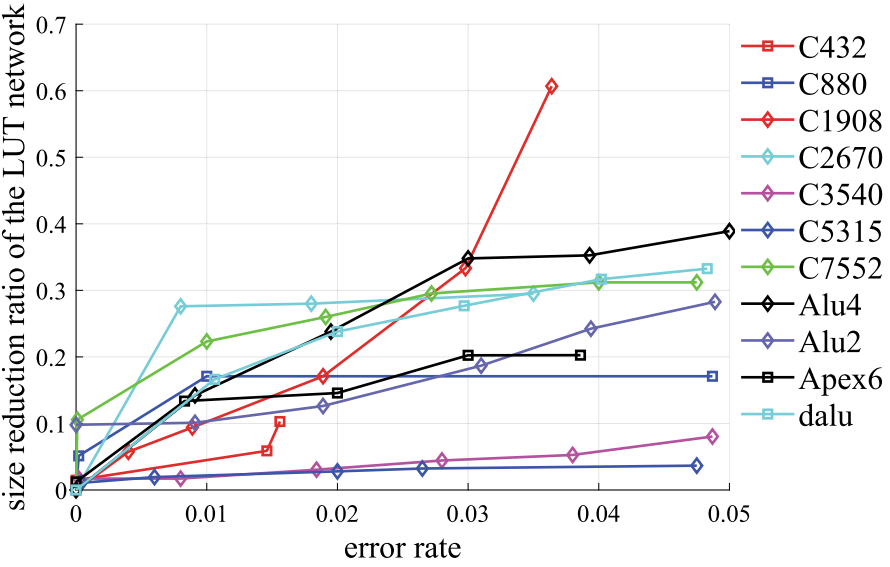

Fig. 7.7 Size reduction ratio vs. error rate plot for the MCNC benchmarks

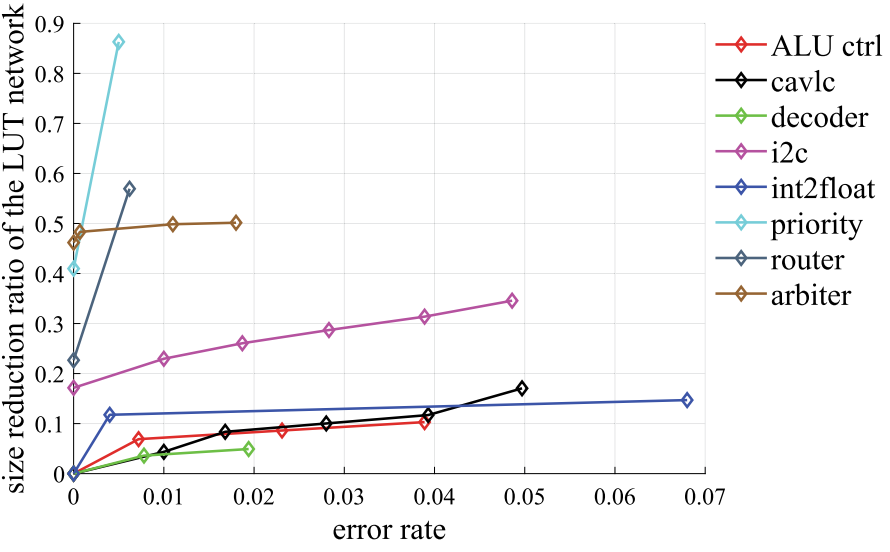

Fig. 7.8 Size reduction ratio vs. error rate plot for the EPFL benchmarks

5.2 Comparison with the Existing ALS Methods for FPGA

We compared our method with the state-of-the-art ALS methods for FPGA [13, 14] using the 11 MCNC circuits. Their synthesis results for the 11 circuits under 5%

ER constraint are also listed in Table 7.1. The columns under "baseline [13, 14]" show the sizes and depths of their baseline input circuits. By comparing the sizes and depths of our baseline inputs and theirs, we find that, on average, our baseline inputs are slightly better than theirs. We believe that it is caused by the different synthesis tools used in the initial FPGA optimization. Note that as our baselines are smaller and faster than theirs, it means that our synthesis task is more challenging than theirs.

Due to the difference in baseline inputs, the SRRs of the methods in [13] and [14] were calculated with their baseline results. The entries in bold highlight the cases where the method of the corresponding column is the best. We note that our method is not as good as the previous methods for some benchmarks, such as `C432`, `C3540`, and `C5315`. There are two reasons for this. One is that some benchmarks (e.g., `C3540`) have few FFCs, causing our method to yield limited improvement. The other is that our baseline inputs for some benchmarks (e.g., `C432` and `C5315`) are much smaller than theirs, making the improvement space of our method limited. Nevertheless, it can be seen that our proposed method achieves more best results than the other two. By comparing the arithmetic mean SRRs listed in Table 7.1, we can also conclude that our method is better than those in [13] and [14] in size reduction. Although our method is area-oriented, it can also effectively reduce the circuit depth. The average depth reduction ratio on the MCNC benchmarks by our method is 14.1%, while that by the method in [14] is 1.5%.

6 Conclusion

In this chapter, we proposed an ALS method for FPGAs. It is based on the novel approximate disjoint and non-disjoint decomposition techniques and the iterative approximate decomposition method, which transforms local LUT subnetworks into approximate ones with the minimum numbers of LUTs. Our experimental results showed that our proposed method is better than the state-of-the-art ALS methods for FPGA.

The current work only considers the error metric as ER. However, it is possible to extend it for other metric like average error magnitude (AEM) by further considering how the local error affects different POs. We will develop method for AEM in our future work.

References

1. Waldrop MM. The chips are down for Moore's law. Nature. 2016;530(7589):144–7.
2. Mittal S. A survey of techniques for approximate computing. ACM Comput Surv. 2016;48(4):62:1–62:33.

3. Shin D, Gupta SK. A new circuit simplification method for error tolerant applications. In: Design, automation & test in Europe conference & exhibition. 2011. pp. 1–6.
4. Venkataramani S, Sabne A, Kozhikkottu V, Roy K, Raghunathan A. SALSA: systematic logic synthesis of approximate circuits. In: Design automation conference. 2012. pp. 796–801.
5. Venkataramani S, Roy K, Raghunathan A. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In: Design, automation & test in Europe conference & exhibition. 2013. pp. 1367–1372.
6. Miao J, Gerstlauer A, Orshansky M. Multi-level approximate logic synthesis under general error constraints. In: International conference on computer-aided design. 2014. pp. 504–510.
7. Vasicek Z, Sekanina L. Evolutionary approach to approximate digital circuits design. IEEE Trans Evol Comput. 2015;19(3):432–44.
8. Chandrasekharan A, Soeken M, D. Große, Drechsler R. Approximation-aware rewriting of AIGs for error tolerant applications. In: International conference on computer-aided design. 2016. pp. 83:1–83:8.
9. Scarabottolo I, Ansaloni G, Pozzi L. Circuit carving: A methodology for the design of approximate hardware. In: Design, automation & test in Europe conference & exhibition. 2018. pp. 545–550.
10. Hashemi S, et al. BLASYS: Approximate logic synthesis using Boolean matrix factorization. In: Design automation conference. 2018. pp. 55:1–55:6.
11. Wu Y, Qian W. ALFANS: Multilevel approximate logic synthesis framework by approximate node simplification. IEEE Trans Comput Aided Des Integr Circ Syst. 2020;39(7):1470–83.
12. Meng C, Qian W, Mishchenko A. ALSRAC: Approximate logic synthesis by resubstitution with approximate care set. In: Design automation conference. 2020. pp. 187:1–187:6.
13. Wu Y, Shen C, Jia Y, Qian W. Approximate logic synthesis for FPGA by wire removal and local function change. In: Asia and South Pacific design automation conference. 2017. pp. 163–9.
14. Liu G, Zhang Z. Statistically certified approximate logic synthesis. In: International conference on computer-aided design. 2017. pp. 344–351.
15. Yao Y, Huang S, Wang C, Wu Y, Qian W. Approximate disjoint bi-decomposition and its application to approximate logic synthesis. In: International conference on computer design. 2017. pp. 517–24.
16. Meng C, Xiang Z, Liu N, Hu Y, Song J, Wang R, Huang R, Qian W. DALTA: A decomposition-based approximate lookup table architecture. In: International conference on computer-aided design. 2021. pp. 1–9.
17. Ashenhurst RL. The decompositions of switching functions. In: International symposium on the theory of switching functions. 1959. pp. 74–116.
18. Curtis HA. A new approach to the design of switching circuits. Van Nostrand; 1962.
19. Shen VS, McKellar AC. An algorithm for the disjunctive decomposition of switching functions. IEEE Trans Comput. 1970;100(3):239–48.
20. Cong J, Ding Y. Flowmap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. IEEE Trans Comput Aided Des Integr Circ Syst. 1994;13(1):1–12.
21. Cong J, Ding Y. Combinational logic synthesis for LUT based field programmable gate arrays. ACM Trans Des Autom Electron Syst. 1996;1(2):145–204.
22. Russell S, Norvig P. Artificial intelligence: a modern approach. Prentice Hall; 2009.
23. Mishchenko A, et al. ABC: A system for sequential synthesis and verification. 2007. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/.