Approximate Logic Synthesis for FPGA by Wire Removal and Local Function Change

Yi Wu, Chuyu Shen, Yi Jia, and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {eejessie, ericolivier, yvonne222, qianwk}@sjtu.edu.cn

Abstract— Approximate computing is a new design paradigm targeting at error-tolerant applications. By allowing a little amount of inaccuracy in the computation, it could significantly reduce circuit area and power consumption. Several logic synthesis methods for approximate computing were proposed recently. However, these methods are mainly aimed at ASIC designs. In this work, we propose a novel approximate logic synthesis method targeting at the FPGA design. We exploit the flexibility of look-up tables and propose a method that combines wire removal and local function change. The experimental results showed that our method produces better results than the state-of-the-art approximate logic synthesis method adapted to FPGA designs. Moreover, it can be combined with the state-of-the-art method to further improve the design quality.

I. Introduction

As modern VLSI designs become more complex and CMOS devices have reached nanoscale, reducing power consumption of the circuits becomes a critical mission in VLSI design. Meanwhile, more and more applications used today exhibit an inherent error tolerance. For example, applications in machine learning, data mining, and pattern recognition demonstrate significant algorithmic resilience to errors, because most of these applications have redundancies in their input data sets and some of them have no golden answers [1]. Given this context, a new computing paradigm called approximating computing was proposed recently which exploits the application-level error tolerance to improve the power consumption and performance of digital circuits [2]. The basic idea of approximate computing is to change the function properly so that a much simpler design can be achieved while still satisfying the error specification. The effectiveness of this idea has been demonstrated in many previous works at different design levels ranging from architecture, logic, to transistor [3–5].

In order to quantify the error, two types of error measurements are used in practice [2]. One is error rate and the other is error magnitude. Error rate is defined as the ratio of input vectors that cause an output error. Error magnitude is generally applied to arithmetic circuits of which the outputs are treated as a numerical value. It is defined as the error of the encoded numerical value. In this work, we focus on approximate computing under error rate constraint.

A number of earlier works in approximate computing studied the manual design of approximate arithmetic circuits, such as adders and multipliers [6–9]. More recently, several approximate logic synthesis (ALS) methods were proposed [4,10–16]. These methods explore the design space in a systematic way to synthesize a low-cost approximate circuit subject to given error constraints. However, all these ALS methods target at the ASIC design. When the design target shifts to FPGA, although these methods can still be applied with proper modifications, they fail to fully exploit the special characteristics of FPGA to further improve the synthesis results. One important feature that can be further exploited is the functional flexibility of the look-up tables (LUTs), which are the basic components of modern FPGA.

In this work, we propose a novel ALS method for FPGA designs under error rate constraint. The main idea of our method exploits the functional flexibility of the LUTs in FPGA, i.e., a k-input LUT can implement any k-input Boolean function. Based on this feature, we propose to derive an approximation of a local sub-network by removing one of its inputs and simultaneously reconfiguring the function. By removing an input of a local sub-network, the number of LUTs needed to cover the network is likely to be reduced. Meanwhile, by properly configuring the function, we can minimize the induced error rate.

The main contributions of our work are as follows.

- We proposed a novel ALS technique which simultaneously removes an input of a local sub-network and changes its function. The technique is highly suitable for FPGA designs, since it exploits the functional flexibility of the LUTs in FPGA.
- We proposed a practical flow to apply the basic technique to the synthesis of FPGA designs. It applies the basic technique to local sub-networks in FPGA and derives candidate transformations. Then it formulates an ILP problem to determine which transformations should be selected to maximize the area reduction.
- We explored the combination of our method with the state-of-the-art method. Experimental results showed that this combined method can further improve the design quality.

The rest of the paper is organized as follows. In Section II, we discuss the related works. In Section III, we introduce some preliminaries that will be used later. In Section IV, we present the basic idea and the detailed techniques of the proposed method. In Section V, we show the overall flow which applies the proposed techniques. In Section VI, we show the experimental results on a set of benchmarks. Finally, in Section VII, we conclude the paper.

II. Related Works

Several previous works have proposed approximate logic synthesis methods. The works [10] and [11] studied ALS for two-level circuits under the error rate constraint. The work [12] further proposed a two-level ALS algorithm considering both the error rate and error magnitude constraints. Several later works [4, 13–16] studied ALS for multi-level circuits, which

978-1-5090-1558-0/17/$31.00 ©2017 IEEE

are more practical designs. In [13], the authors derived an approximate circuit by injecting stuck-at faults into the original circuits. In [4], the authors introduced a quality constraint circuit to convert the ALS problem under error magnitude constraint into a conventional logic synthesis problem, which can be solved by existing logic synthesis tools. In [14], the authors proposed an ALS method for multi-level circuits that works under both the error rate and the error magnitude constraints. They first solved the error-magnitude-constrained problem by formulating it as a Boolean relation minimization problem. Then they enforced the error rate constraint by iteratively recovering some incorrect outputs. In [15], the authors proposed a method based on identifying signal pairs that take the same value with high probability and substituting one for the other to reduce area. In [16], the authors proposed an efficient method to synthesize multi-level approximate circuits under the error rate constraint. Their basic idea is to pick nodes in a Boolean network and shrink them by approximating their factored-form expressions. A knapsack problem was formulated to pick multiple nodes for shrinking simultaneously.

Those previous methods for synthesizing multi-level approximate circuits are developed for ASIC designs. Some of them operate on different circuit representations than a LUT network and hence are not applicable to FPGA designs. For example, the method in [16] manipulates the factored-form expressions of the nodes in an abstract Boolean logic network, which is significantly different from a network composed of LUTs. Some other methods can be modified to handle LUT-based FPGA designs. For example, the approach in [15] only needs to find signal pairs with high similarity in a circuit, which is also suitable for a LUT network. However, they fail to capture the special properties of FPGAs, for example, the functional flexibility of a LUT. As a result, we believe there still exists room for further improvement for the ALS algorithms targeting at FPGAs. This work is the first attempt to explore the extra flexibility of FPGA to further improve the quality of the approximate circuits. The way we explore the additional flexibility is to properly remove inputs and reconfigure the Boolean function.

III. Preliminaries

In this section, we will briefly introduce the basic concepts related to LUT-based FPGAs. An FPGA is an off-the-shelf VLSI chip consisting of programmable logic elements, programmable I/O elements, and programmable routing elements [17]. The most widely used FPGA designs are composed of look-up tables (LUTs). A k-input LUT (called k-LUT for short) can implement any single-output Boolean function of up to k variables. For many FPGA architectures, k is between 3 and 6 [18].

To synthesize an FPGA design, a technology-independent synthesis procedure is first called to generate a Boolean logic network for the target function. Then, a technology mapping procedure is called to map the Boolean logic network into a LUT network.

A LUT network is a directed acyclic graph (DAG), where the nodes are primary inputs/outputs (PI/POs) or k-input LUTs and each directed edge (v, u) indicates that the output of node v is an input of node u. Given a network N, the set of distinct nodes that input to a sub-network H is denoted as $input(H)$. The size of $input(H)$ is denoted as $|input(H)|$.

Given a node v in the network N, a cone of v, denoted as C_v, is a sub-network of N consisting of v and some of its non-PI predecessors such that for any node $w \in C_v$, there is a path from w to v that lies entirely in C_v. 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 (i.e., they converge to the root). For example, Fig. 1 shows a network with i_1, i_2, i_3 as primary inputs. Sub-networks H_1, H_2 and H_3 are three FFCs of node a, where $input(H_1) = \{i_1, i_2, e\}$, $input(H_2) = \{i_1, c\}$, and $input(H_3) = \{b, d, e\}$. Different FFCs of one node always overlap with each other because they all contain the node itself.

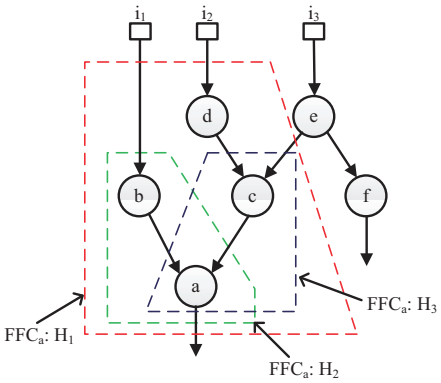

Fig. 1. An illustration of FFC.

IV. Proposed Method

In this section, we will describe the basic idea of our method first, followed by the details of the proposed techniques to approximate an FPGA design.

A. Basic Idea

As introduced in Section III, the combinational logic synthesis for FPGAs includes two steps: the technology-independent synthesis step and the technology mapping step. The first step often aims for intuitively good Boolean networks, such as those with a small number of gates, small gate input size, and small depth. The rationale of our approach is based on the observation that if the number of inputs of a Boolean network is reduced, the number of LUTs needed to cover the network is also likely to be reduced.

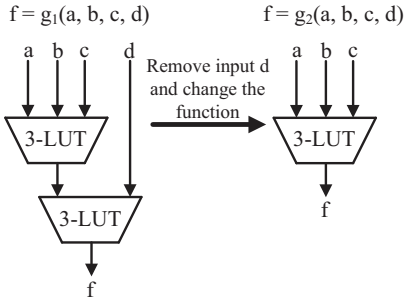

Fig. 2. An example showing the basic idea of the proposed approach.

Fig. 2 shows a small example which illustrates the basic idea of our approach. Originally, f is a function of 4 inputs. The mapped LUT network is composed of two 3-LUTs as shown in the left part of the figure. If it is possible to find a close approximation to the original function which has one input fewer, then only one LUT is needed to realize this approximate function, as shown in the right part of Fig 2. In other words, by wire re-

moval and function change, we may find a close approximation to the original FPGA design with fewer number of LUTs.

B. Techniques for Approximate Logic Synthesis for FPGA

Theoretically, the basic idea introduced in Section IV-A can be applied either to the entire network or to a local sub-network. However, it is not practical to approximate the entire network by removing some of its primary inputs and changing its function since real FPGA designs are usually very large. Therefore, we choose to implement the basic idea in a local way.

Specifically, we target at FFCs of a LUT network and apply the basic idea to each FFC. In a LUT network, each LUT is treated as a node. A node usually has more than one FFC rooted at that node. Given an FFC of one node, we consider all possible cases of removing one of its inputs. For each case, we also modify the function so that the resultant error rate of the whole design is as small as possible. Since the number of inputs is reduced by one, the number of LUTs needed to realize the new function may be reduced. If this happens, this modification is a good candidate local approximation.

Next, we use an example to illustrate the details of the proposed technique. Suppose that we have an FFC with 3 inputs I_1, I_2, and I_3 and one output O. Its truth table is shown in Table I(a). Now, assuming that we want to remove the input I_3, then how should we determine the new function for the FFC? In other words, what will the new truth table with only I_1 and I_2 as inputs be? The answer is that we derive the new truth table from the original one under the criteria that the increased error rate due to this modification is minimized. In this example, when $\{I_1I_2\}$ takes the pattern 00, the output O is 1 no matter I_3 is 0 or 1. Therefore, for the new truth table, we set output O as 1 for input pattern 00. Likewise, when $\{I_1I_2\}$ takes the value 11, we set output O as 0. However, for input pattern $\{I_1I_2\} = 01$ or 10, the output O takes different values for different values of I_3. In such cases, the error rate should be taken into account.

TABLE I
ORIGINAL AND MODIFIED TRUTH TABLES OF AN FFC.

(a) Original truth table of an FFC

I_1	I_2	I_3	O
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(b) Modified truth table of an FFC.

I_1	I_2	O
0	0	1
0	1	0
1	0	1
1	1	0

When the local Boolean function of an FFC is changed, it may cause errors at the primary outputs (POs). However, it is very time-consuming to obtain the actual error rate observed at the POs. According to [16], we can use the error rate observed at the output of the FFC as an estimate of the actual error rate. This error rate is called *apparent error rate* [16]. It gives an upper bound to the actual error rate. It can be easily calculated as the sum of the probabilities of the local input patterns whose output values are changed. For example, if we change the output values for the input patterns $\{I_1I_2I_3\} = 011$ and 101 of the FFC in the above example, then the apparent error rate is $Pr(011) + Pr(101)$, where $Pr(X)$ denotes the probabilities of a pattern X. Due to these properties, we use apparent error rate in this work as an estimate to the actual error rate. In the following, unless otherwise specified, when we say error rate, it refers to the apparent error rate.

For the above example, suppose $Pr(011) \leq Pr(010)$ and $Pr(101) \leq Pr(100)$. In order to minimize the total error rate, the values of input patterns $\{I_1I_2\} = 01$ and 10 in the new truth table should be set as the original output values of the patterns $\{I_1I_2I_3\} = 010$ and 100, respectively. The general rule is that we always *flip* the output value of the input pattern whose probability is *smaller*. This guarantees that the total induced error rate is minimized. The modified truth table is shown in Table I(b). The total error rate induced is equal to $Pr(011) + Pr(101)$.

In general case, suppose an FFC under consideration has n input variables $x_1, x_2, \cdots, x_n$ and its function is $F(x_1, x_2, \ldots, x_n)$. Assume we want to remove the input x_i. Then, the technique introduced above requires us to find all input patterns $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ such that $F(x_1, \ldots, x_{i-1}, 0, x_{i+1}, x_n) \neq F(x_1, \ldots, x_{i-1}, 1, x_{i+1}, x_n)$. This is equivalent to finding the input patterns such that

$$F(x_1, \ldots, x_{i-1}, 0, x_{i+1}, x_n) \oplus F(x_1, \ldots, x_{i-1}, 1, x_{i+1}, x_n) = 1.$$

The left-hand side of the above equation is just the *Boolean difference* of F with respect to x_i, which tells us how the function F changes with input x_i. In our implementation, we use binary decision diagram (BDD) [19] to obtain the Boolean difference and then derive the input patterns that have different output values.

For the above technique, we also need to obtain the probability of any local input pattern. This is achieved by logic simulation and counting the frequency of each input pattern.

If the total error rate is less than or equal to the allowed error rate, we will check how many LUTs can be reduced when the function of the FFC is replaced by the closest approximation. Specifically, we apply the logic synthesis tool *ABC* [20] to map the modified truth table into a LUT network. If the new LUT network has fewer LUTs than the original LUT network, we will list it as a *candidate transformation*. For each FFC, we will consider all cases of removing one input and obtain all candidate transformations.

We call the above type of transformation obtained by removing an input of an FFC and then deriving the closest approximation the **FFC-type** transformation. Besides this type of transformation, we also include **constant-type** transformation, which approximates a single node in the network as a constant 0 or a constant 1. Indeed, when the signal probability p of the output of a node satisfies that $p \leq T$ (or $1 - p \leq T$), where T is the error rate threshold, the approximation of the node function as a constant 0 (or a constant 1) is also a feasible approximation for that node, since the increased error rate by this approximation is below T. Meanwhile, this approximation could lead to further removal and simplification of some other nodes. Therefore, when the signal probability p of the output of a node satisfies that $p \leq T$ (or $1 - p \leq T$), we will also include the approximation of that node by a constant 0 (or a constant 1) as a candidate transformation. The LUT number reduction of a constant-type transformation is obtained by temporarily applying it to the orignal network to check how many LUTs can be removed through constant propagation. For an FFC-type transformation, we define its *root node* as the root of that FFC. For a constant-type transformation, we define its *root node* as the node which we reduce to a constant value.

V. ALGORITHM

Using the techniques introduced in Section IV-B, we propose an algorithm for synthesizing an optimal approximate

FPGA design under the error rate constraint. The algorithm is iterative. In each iteration, we pick multiple transformations among all candidate FFC-type and constant-type transformations and apply them simultaneously to the intermediate approximate design derived from the previous iteration. In the following, we will first discuss how we decide which set of transformations to select. Then, we will show some speed-up techniques. Finally, we summarize the overall flow of the proposed algorithm.

A. Selecting Multiple Transformations

We formulate the selection of multiple transformations as an integer linear programming (ILP) problem. Solving this problem helps us decide which transformations should be selected simultaneously such that the number of LUTs saved can be maximized while the total error rate is still within the threshold.

Specifically, we first check each node in a network and find all candidate FFC-type and constant-type transformations. A *candidate transformation* is one that reduces the number of LUTs and at the same time has error rate below the error rate margin. They are obtained by the techniques introduced in Section IV-B.

Suppose there are m candidate transformations $t_1, t_2, \cdots, t_m$. For each candidate transformation t_i, we associate a binary variable v_i with it. If in the final solution $v_i = 1$, then the transformation t_i is selected. Otherwise, it is not chosen. For each transformation t_i, assume its error rate is e_i and its LUT number reduction is a_i. The objective is to maximize the total LUT number reduction of all the selected transformations, which is calculated as

$$\sum_{i=1}^{m} a_i v_i. \tag{1}$$

Next, we discuss the constraints on the v_i's. In order to perform the selected transformations simultaneously, we require the affected regions of two transformations do not overlap. The affected region of a transformation is defined as follows.

Definition 1
For a constant-type transformation of a node n, the affected region is n itself. For an FFC-type transformation, the affected region is the FFC on which we derive the transformation.

Notice that if we apply a constant-type transformation of a node n, besides removing the node n, it may also induce some changes to its surrounding nodes by constant propagation. However, the changes depend on the other transformations we simultaneously pick. Without knowing the other selected transformations yet, we can only guarantee that the node n will be modified. In a sense, the affected region we define here is a region that is guaranteed to be modified by applying a transformation.

If the affected regions of two transformations t_i and t_j overlap, then they cannot be selected simultaneously. Given that the affected region of a transformation is either an FFC or a single node, there are two situations where the affected regions of two transformations t_i and t_j overlap. The first situation is shown in Fig. 3(a), in which the root nodes of the two transformations are the same. The second situation is shown in Fig. 3(b), in which the root node of the transformation t_i is not the same node as that of the transformation t_j, but is inside the affected region of t_j. The reason that we cannot select these overlapping transformations together is that after one transformation is performed, the affected region of the other transformation is altered. Take the case of Fig. 3(a) as an example. After the transformation t_i is applied, the affected region of t_i is changed into a completely new sub-network, which also modifies the original affected region of t_j. Thus, the original transformation t_j cannot be performed now.

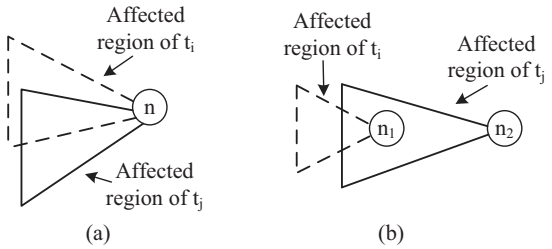

Fig. 3. Illustration for cases where the affected regions of two transformations t_i and t_j overlap.

To avoid the simultaneous selection of two transformations t_i and t_j with overlapping affected regions, we add the following constraint

$$v_i + v_j \leq 1. \tag{2}$$

To obtain all the constraints, after a new candidate transformation is found, we compare it with each candidate transformation already obtained to check whether the two have overlap.

Another constraint on the final choice is that the actual error rate caused by applying all the selected transformations should be within the error rate threshold t. As shown in [16], if the affected regions of a set of transformations do not overlap, then the sum of their apparent error rates is an upper bound on the actual error rate caused by applying these transformations. Therefore, we include the following constraint on the error rate

$$\sum_{i=1}^{m} e_i v_i \leq t, \tag{3}$$

where e_i is the apparent error rate of the transformation t_i. If the above constraint is satisfied, the actual error rate will also be below the error rate threshold.

To summarize, our ILP formulation maximizes the target function shown in Eq. (1) under the set of constraints in the form of Eq. (2) on all pairs of overlapping transformations and the constraint shown in Eq. (3).

B. Speed-up Techniques

To improve the efficiency of the proposed algorithm, when we search for the candidate FFC-type transformations, we only consider FFCs with number of inputs within a range. If the FPGA design is built with k-LUTs, then the lower bound of the range is chosen as $k + 1$. The reason that we do not consider FFCs with fewer than $k+1$ inputs is that such FFCs are covered by only one LUT. In this case, removing one input of this FFC leads to no reduction in the number of LUTs except that this FFC can be reduced to a constant. The latter case is covered by the constant-type transformation. In contrast, the upper bound of the range can be set freely. Usually, a larger upper bound will produce a result with better quality while consuming more runtime since the number of candidate transformations increases and a larger design space is explored. A proper value for the upper bound can be determined based on the requirement for the design quality and runtime.

We also observed that when the number of overlapping transformations is large, it will take a long time to solve the ILP formulation. To overcome this issue, we propose to ignore a num-

ber of low-score FFC-type transformations for large benchmarks. Specifically, we define the score of a candidate transformation as a/e, where a represents the number of LUTs saved and e denotes its error rate. For each node n, we order all candidate FFC-type transformations of node n by their scores and discard the transformations whose scores are the lowest $q\%$. (In our experiment, we choose $q = 40$.) This is reasonable since the FFC-type transformations of the same root node overlap with each other and the ones with lower scores are less likely to be selected.

Another speed-up technique is that when checking an FFC for possible transformations, we can first obtain the probabilities of all of its input patterns. If all of these probabilities exceed the error rate threshold, we can immediately conclude that we cannot derive any candidate transformations from this FFC through function change.

C. Flow of the Proposed Algorithm

The flow of the entire algorithm is shown in Algorithm 1. Its inputs include a LUT network C and an error rate threshold T. The approximate network is denoted as C', which is initially set as the input network C. The variable ER records the error rate of the network C' at the start of each iteration. ER is 0 initially.

Algorithm 1 The proposed algorithm.

```
1: inputs: an input LUT network C and a threshold T on error rate.
2: outputs: an approximate LUT network C' with minimal LUT
   number and error rate ≤ T.
3: initialize: C' ⇐ C; current error rate ER ⇐ 0;
4: while true do
5:    error rate margin t ⇐ T − ER; trans_set ⇐ ∅;
6:    for each node n in the network C' do
7:       if n's signal probability p ≤ t or 1 − p ≤ t then
8:          put constant-type transformation of n into trans_set;
9:       for each FFC c of node n such that k + 1 ≤ |input(c)| ≤
         l do {l limits the maximal input size of the FFCs we will
         consider.}
10:         for each input x_i of c do
11:            obtain FFC-type transformation tr by removing input
               x_i;
12:            if tr.error_rate ≤ t && tr has fewer LUTs && ap-
               plying tr does not increase the delay of C' then
13:               put tr into trans_set;
14:   formulate the ILP problem on all transformations in trans_set
      under the error rate margin t;
15:   solve the ILP problem;
16:   if there is no transformation selected then
17:      break;
18:   apply the selected transformations to the network C';
19:   update ER by running logic simulation;
20: return C';
```

The algorithm has a number of iterations. In each iteration, it selects and applies a number of candidate transformations to the latest approximate network to derive a new approximate network. It terminates when in an iteration, there is no transformation selected (Lines 16–17), so there is no update to the approximate network from the previous iteration. The network C' in the last iteration is returned as the final approximate network (Line 20).

During each iteration, we first set the current error rate margin t as $T - ER$ (Line 5). Then we find all the candidate transformations and store them in the set $trans\_set$ (Lines 6–13). To achieve this, we iterate over all the nodes in the current network C' (Line 6). For each node n, we first check whether it has a candidate constant-type transformation (Line 7). If so, the transformation is put into the set $trans\_set$ (Line 8). Then, all FFCs of node n whose number of inputs are in the range $[k + 1, l]$ are examined for FFC-type transformations (Line 9), where l is a parameter limiting the maximal input size of the FFCs we will consider. For each input x_i of each FFC c, if the transformation obtained by removing x_i satisfies that (1) its error rate is smaller than the error rate margin t, (2) it has fewer LUTs than the original FFC, and (3) applying the transformation does not increase the delay of C', then it will be included into the set $trans\_set$ (Lines 12–13). Note that in our implementation, we also require a candidate transformation does not increase the original delay if it is applied. To check this condition, we first temporarily apply the transformation to the network and then check whether the delay of the network increases. In our implementation, the delay of a network is reported by the logic synthesis tool *ABC* [20]. After we obtain all the candidate transformations, we set up the ILP problem as discussed in Section V-A (Line 14). For this ILP, the error rate threshold is set as the current error rate margin t. Then, the GNU Linear Programming Kit (GLPK) package [21] is called to solve the ILP problem (Line 15). The output of the solver tells which transformations should be selected. If there is no transformation selected, the loop terminates (Lines 16–17). Otherwise, we apply the selected transformations to derive a new approximate network (Line 18). At the end of each iteration, we calculate the actual error rate ER of the current network C' by running logic simulation and counting the frequency that the POs are incorrect (Line 19). Note that we also use this simulation result to obtain the probabilities of local input patterns which are needed in the next iteration. Compared with searching candidate transformations, running logic simulation is much faster.

By our error rate constraint shown in Eq. (3), the sum of the error rates of the selected transformations is smaller than the current error rate margin. Furthermore, as we discussed in Section V-A, the actual error rate is smaller than that sum. Therefore, the actual error rate is guaranteed to be smaller than the error rate margin t. As a result, the final obtained approximate design has its actual error rate below the given threshold T. On the other hand, however, we do not need to worry that the actual error rate of the final approximate design would be much less than the threshold. The reason is that we always check the actual error rate at the end of each iteration through simulation and the iterations will not stop until the threshold is reached, unless no candidate transformations are applicable.

For each node, given that we only consider FFC with bounded input size, it takes constant time to find all the candidate transformations of the node. Besides, according to our experiments, formulating and solving the ILP problem take much less time than searching candidate transformations. Therefore, the runtime for one iteration of the algorithm is linear to the number of nodes in the current design.

VI. Experimental Results

We implemented the proposed algorithm in *C++*. To demonstrate its effectiveness, we conducted two experiments on MCNC benchmarks [22]. The FPGA designs were obtained by executing the FPGA mapping command *if* in the logic synthesis tool *ABC* [20] multiple times on the original Boolean networks until there is no further improvement in the number of LUTs. We considered FPGA technology using 4-LUTs. The information of the FPGA designs used as benchmarks is shown in Table II. The "#LUTs" and "level" columns list the numbers of LUTs and the levels of the FPGA designs. All the experiments were conducted on a virtual machine running Linux operating system with 1GB memory. The host machine is a 3.1 GHz desktop.

TABLE II
BENCHMARK INFORMATION.

Name	I/O	Function	#LUTs	level
c432	36/7	Priority Decoder	97	10
c880	60/26	8-bit ALU	128	8
c1908	33/25	16-bit SEC/DED circuit	122	9
c2670	233/140	12-bit ALU and controller	295	7
c3540	50/22	8-bit ALU	346	12
c5315	178/123	9-bit ALU	503	9
c7552	207/108	32-bit adder/comparator	593	8
alu4	14/8	ALU	710	7
alu2	10/6	ALU	160	12
apex6	135/99	Logic	253	6
dalu	75/16	Dedicated ALU	425	11

In our implementation, we used the Verilog logic simulation tool Synopsys VCS [23] to do logic simulation. We assumed all PI patterns were of equal probability. Each run of logic simulation applied 10000 randomly generated PI vectors. For comparison, we also implemented the state-of-the-art method SASIMI [15] and applied it to FPGA designs. Since in our method, we only consider candidate transformations that do not increase the delay of the original LUT network, the approximate designs produced by our method have delays smaller than or equal to the original designs.

In the first experiment, we ran the proposed algorithm and the SASIMI algorithm [15] on FPGA designs. In our proposed algorithm, there is a parameter l limiting the maximal input size of the FFCs we will consider. For simplicity, we call the parameter l *input limit*. We set l as 7, 8 and 9, respectively, to study its effect on the trade-off between the design quality and the runtime.

TABLE III
LUT NUMBER REDUCTION AND RUNTIME COMPARISONS BETWEEN OUR APPROACH AND THE SASIMI APPROACH [15] ON FPGA DESIGNS.

	input_limit(9)		input_limit(8)		input_limit(7)		SASIMI	
circuit	#LUTs ratio	time/s	#LUTs ratio	time/s	#LUTs ratio	time/s	#LUTs ratio	time/s
c432	0.86	66	0.86	51	0.86	41	0.97	4
c880	0.83	92	0.84	61	0.86	50	0.92	24
c1908	0.64	52	0.65	35	0.67	25	0.68	50
c2670	0.83	139	0.84	85	0.85	61	0.87	31
c3540	0.95	259	0.95	179	0.96	164	0.95	144
c5315	0.94	264	0.94	168	0.95	131	0.97	197
c7552	0.86	430	0.88	280	0.89	170	0.89	522
alu4	0.78	2325	0.79	1500	0.80	1280	0.83	996
alu2	0.89	247	0.9	112	0.91	91	0.96	27
apex6	0.80	166	0.81	111	0.82	107	0.86	78
dalu	0.83	811	0.84	536	0.86	457	0.82	471
GEOMEAN	0.83	228	0.84	149	0.85	119	0.88	87

Comparisons of the LUT number reduction and runtime between the proposed method and the SASIMI on FPGA designs are presented in Table III. The "input_limit(9)", "input_limit(8)" and "input_limit(7)" columns show the results of our approach with $l = 9$, 8 and 7, respectively. The columns titled "SASIMI" presents the results of the SASIMI approach. For each benchmark, we ran the experiments for three different error rate thresholds: 1%, 3%, and 5%. Due to the randomness in the logic simulation, for each error rate threshold, the experiment was run five times and the average LUT number and runtime were obtained. Then, we calculated the ratio of the average LUT number of the obtained approximate design to the LUT number of the original design. Finally, the average ratio on the LUT number and the average runtime over the three error rates were computed and listed in the "#LUTs ratio" column and "time/s" column in Table III, respectively.

As shown in Table III, our approach could produce smaller approximate circuits than the SASIMI approach [15] for almost all benchmarks. For the proposed method, the approximate circuits obtained with a larger input limit l usually have fewer numbers of LUTs, while the runtime for a larger l is longer. The last row of Table III gives the geomean of the ratios on LUT number and geomean of the runtimes over all benchmarks for the proposed method and SASIMI. We can see that the geomean of ratios on LUT number for all the three l values are better than that of SASIMI. The runtime of our method is $1.3\times \sim 2.6\times$ more than SASIMI.

To more clearly compare the quality of the proposed method and SASIMI, we plot the ratio of the LUT number of the approximate circuit obtained by the proposed method to that obtained by SASIMI for all benchmarks in Fig. 4. The figure includes the results of our method with input limit $l = 7, 8, 9$. For all the benchmarks except `c3540` and `dalu`, our method with $l = 9$ could produce approximate designs with LUT numbers equal to 87%∼97% of that produced by SASIMI.

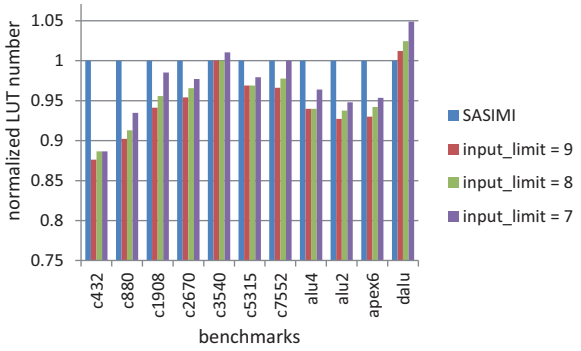

Fig. 4. Ratios of the LUT number by the proposed method to the LUT number by SASIMI [15].

Fig. 5 plots how the reduction ratio on LUT number changes with the error rate by applying our method with input limit $l = 9$. The reduction ratio on LUT number was calculated as the reduced number of LUTs by our method to the LUT number of the original design. As we can see, with the increase of error rate, the reduction ratio on LUT number for each benchmark also increases. When the error rate is about 1%, the reduction ratio for each benchmark is between 5% and 15%. When the error rate is raised to 3%, this value is in the range from 6% to 40%. Finally, for error rate of 5%, the reduction ratios for most benchmarks are between 15% and 60%.

In the second experiment, we tested whether the proposed method can be combined with SASIMI to further improve the design quality. As we observed, for several benchmarks, the error rates of the approximate designs obtained by SASIMI are still far away from the given error rate thresholds. For example, for benchmark `c2670` and the specified error rate threshold of 3%, the final error rate of the approximate design obtained by SASIMI is only 1.64%, which leaves an error rate margin of 1.36%. In this case, we can further apply our method to the approximate design returned by SASIMI by assigning the error rate threshold as 1.36%. We did this experiment on five benchmarks including `c2670`, `c3540`, `c5315`, `apex6`, and `dalu` for three different error rate thresholds of 1%, 3%, and 5%. We chose these benchmarks because the error rates of the approximate circuits obtained by SASIMI are still far away from the given error rate thresholds. For our method, we chose

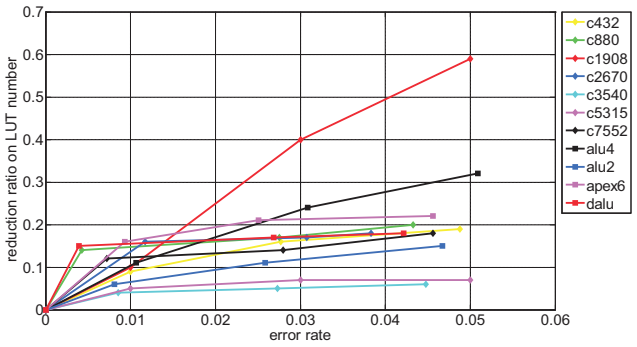

Fig. 5. Reduction ratios on LUT numbers produced by our proposed method versus error rates for all benchmarks.

the input limit $l = 9$. For each error rate threshold, the experiments were performed five times. The average results are shown in Table IV. The first column shows the names of the benchmarks along with the numbers of LUTs in the original FPGA designs. The "SASIMI_only" column lists the numbers of LUTs in the approximate designs obtained by SASIMI, while the "SASIMI+Ours" column lists the numbers of LUTs in the final approximate designs obtained by further applying our method. From the results we can see that the design qualities are improved by the combination of the two methods. This demonstrates that our method is able to search a different solution space and exploit different optimization opportunities compared with SASIMI. In other words, our method can be seen as a good complement to SASIMI.

TABLE IV
EXPERIMENTAL RESULTS FOR THE COMBINATION OF SASIMI AND OUR PROPOSED METHOD.

benchmark	error rate threshold	SASIMI_only	SASIMI+Ours
c2670 (295)	0.01	259	251
	0.03	252	244
	0.05	250	240
c3540 (346)	0.01	337	333
	0.03	330	325
	0.05	322	316
c5315 (505)	0.01	491	479
	0.03	489	476
	0.05	487	467
apex6 (253)	0.01	225	215
	0.03	216	211
	0.05	212	209
dalu (425)	0.01	353	342
	0.03	348	319
	0.05	339	311

VII. CONCLUSION

In this paper, we proposed a novel approach for approximate logic synthesis targeting at LUT-based FPGA designs under the error rate constraint. We exploited the functional flexibility of LUTs and proposed a technique that removes one input of a fanout-free cone (FFC) and configures the function of the FFC so that the induced error rate is minimal. By applying the technique to some FFCs in the LUT network, we can obtain a set of candidate transformations. We further proposed an algorithm which selects a set of non-overlapping candidate transformations to apply simultaneously. The selection problem was formulated as an integer linear programming problem. We applied our algorithm on MCNC benchmarks and the experimental results showed that it could derive more area-efficient approximate designs than the state-of-the-art method SASIMI [15]. In addition, the proposed method can be combined with SASIMI to further improve the design quality in some cases.

ACKNOWLEDGMENTS

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

REFERENCES

[1] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC'13*, pp. 1–9.

[2] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *ETS'13*, pp. 1–6.

[3] M. Imani, A. Rahimi, and T. Rosing, "Resistive configurable associative memory for approximate computing," in *DATE'16*, pp. 1327–1332.

[4] S. Venkataramani, A. Sabne, V. Kozhikkottu *et al.*, "SALSA: Systematic logic synthesis of approximate circuits," in *DAC'12*, 2012, pp. 796–801.

[5] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[6] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *ISIC'09*, pp. 69–72.

[7] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC'12*, pp. 820–825.

[8] M. E. P. Kulkarni, P. Gupta, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSID'11*, pp. 346–351.

[9] C. Liu, J. Han, and F. Lowbardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *DATE'14*, pp. 1–4.

[10] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE'10*, pp. 957–960.

[11] C. Zou, W. Qian, and J. Han, "DPALS: A dynamic programming-based algorithm for two-level approximate logic synthesis," in *ASICON'15*, pp. 1–4.

[12] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD'13*, pp. 779–786.

[13] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *DATE'11*, pp. 1–6.

[14] J. Miao, A. Gerstlauer, and M. Orshansky, "Multi-level approximate logic synthesis under general error constraints," in *ICCAD'14*, pp. 504–510.

[15] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *DATE'13*, pp. 796–801.

[16] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *DAC'16*.

[17] J. Cong and Z. 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.

[18] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *ICCAD'07*, pp. 354–361.

[19] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. c-35, no. 8, pp. 677–691, 1986.

[20] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," 2007. [Online]. Available: http://www.eecs.berkeley.edu/ alanmi/abc/

[21] GLPK, http://www.gnu.org/software/glpk/.

[22] S. Yang, "Logic synthesis and optimization benchmarks," Microelectronics Center of North Carolina, Tech. Rep., 1991.

[23] Synopsys, http://www.synopsys.com/.