ALSRAC: Approximate Logic Synthesis by Resubstitution with Approximate Care Set

Chang Meng[1], Weikang Qian[1,2,3], Alan Mishchenko[4]
[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China
[2]MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, Shanghai, China
[3]State Key Laboratory of ASIC & System, Fudan University, Shanghai, China
[4]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
Email: changmeng@sjtu.edu.cn, qianwk@sjtu.edu.cn, alanmi@berkeley.edu

Abstract—**Approximate computing is an emerging design technique for error-resilient applications. It improves circuit area, power, and delay at the cost of introducing some errors. Approximate logic synthesis (ALS) is an automatic process to produce approximate circuits. This paper proposes approximate resubstitution with approximate care set and uses it to build a simulation-based ALS flow. The experimental results demonstrate that the proposed method saves 7%–18% area compared to state-of-the-art methods. The code of ALSRAC is made open-source.**

Index Terms—**Approximate Logic Synthesis, Approximate Computing, Resubstitution, Approximate Care Set**

I. Introduction

As power consumption in digital systems grows rapidly, energy efficiency has become a crucial concern [1]. Fortunately, many applications that are widely used today, such as image processing, data mining, and machine learning, exhibit error resilience. This feature leads to a new paradigm for designing energy-efficient digital systems, known as **approximate computing**. The key idea is to modify the function implemented by the system, which can be equivalently viewed as introducing some errors into the system. If carefully managed errors are introduced, the application-level quality is almost unaffected, while the hardware cost and power consumption of the system can be reduced dramatically.

Many approaches have been developed to generate approximate circuits. They can be divided into two categories: manual design and **approximate logic synthesis** (**ALS**). Manual design is popular for arithmetic circuits such as adders [2] and multipliers [3], while ALS targets general circuits. Given an original circuit and some error constraints, an ALS method automatically synthesizes an approximate circuit of high quality satisfying these constraints. Typical error constraints include error rate and error distance constraints, while typical quality measures include circuit area, delay, and power.

This paper focuses on the ALS for multi-level combinational circuits. Most ALS approaches can be classified into two categories: stochastic methods and deterministic methods.

A stochastic ALS method is characterized by uncertainties when simplifying the circuit. For example, Vasicek *et al.* apply the Cartesian genetic programming to randomly change local circuit structure [4]. Liu *et al.* utilize the Markov chain Monte Carlo method to probabilistically perform circuit modifications [5]. The stochastic approaches often lead to quite different results in each run.

A deterministic ALS method is featured with deterministic modifications to the circuit. Some deterministic methods transform an ALS problem into a traditional logic synthesis problem and utilize logic synthesis tools to solve it. For example, SALSA [6] and the method in [7] identify approximate external don't-cares (EXDCs) from the maximum error distance constraint. Then, ALS is converted into a logic synthesis problem with EXDCs. However, they are unscalable for large circuits, since the number of EXDCs increases exponentially with the number of **primary inputs** (**PIs**).

Most deterministic methods modify local structures in the circuit. Such modification is called a **local approximate change** (**LAC**). Many types of LACs tend to have two drawbacks: they are either **too local** or **too coarse**. A LAC is too local if it approximates the node's function using only nearby nodes, such as direct fanins. More distant nodes are not used to express the approximate function, which may lose opportunities for better approximation. For instance, Wu and Qian introduce a LAC for Boolean networks, which removes literals from the Boolean expression of each node in the network [8]. In this case, the approximate function is expressed with the node fanins. Yao *et al.* approximate a maximum fanout-free cone (MFFC) in the circuit by a tree of gates obtained through approximate disjoint bi-decomposition [9]. The new function is represented by the inputs of the MFFC. On the other hand, a LAC is too coarse if it fails to closely approximate the original function. Thus, large errors may occur. For instance, Shin and Gupta propose to replace a gate by a constant zero or one [10]. Chandrasekharan *et al.* rewrite the local critical cuts in an AND-inverter graph with constant zero [11]. Obviously, such constant substitutions produce large errors in the circuit. Venkataramani *et al.* propose SASIMI where a LAC substitutes a node by another node with a similar function [12]. Su *et al.* further improve SASIMI by an accurate and efficient batch error estimation approach [13]. Yet, such a single-input substitution may also introduce large errors, since a function usually depends on multiple inputs and the expressive power of a single input is limited.

To address the above challenges of deterministic ALS methods based on local changes, we attempt to find a LAC that replaces a node function with a multi-input function. It is neither too local if remote nodes are selected to express the new function, nor too coarse due to the rich expression power of multi-input functions. Notice that in traditional logic synthesis, such a replacement is called a **resubstitution**. Previous work [14] generates resubstitutions using the care set of a node. In this work, we propose to generate **approximate resubstitutions** using an approximate care set.

Our main contributions are as follows:

- We propose to approximate the care set using logic simulation. For scalability, the care patterns are expressed in terms of internal nodes instead of the PIs.
- We propose to generate approximate resubstitutions by computing **irredundant sums-of-products** (**ISOPs**) using approximate care patterns. This novel type of LAC exploits distant signals to form an approximation and has strong expressive power.
- We propose **ALSRAC**, an <u>ALS</u> flow by <u>r</u>esubstitution with <u>a</u>pproximate <u>c</u>are set. It is an efficient simulation-only logic synthesis flow that does not rely on complex Boolean manipulation engines, such as **satisfiability** (**SAT**) and **binary decision**

This work is supported in part by the State Key Laboratory of ASIC & System Open Research Grant 2019KF004, the National Natural Science Foundation of China (NSFC) Grant 61574089, and SRC contract 2867.001. Corresponding author: Weikang Qian.

diagram (**BDD**).

ALSRAC is applicable to statistical error metrics such as error rate and mean error distance. The experimental results show that it improves the quality of approximate circuits significantly on various benchmarks and error metrics. It achieves 7%–18% more area savings than the state-of-the-art methods. It also improves the circuit delays and has much shorter runtime for most circuits. The code of ALSRAC is made open-source at https://github.com/SJTU-ECTL/ALSRAC.

The remainder of this paper is organized as follows. Section II introduces the preliminaries. Section III elaborates the proposed LAC and the ALSRAC methodology. The experimental results are presented in Section IV. Finally, Section V concludes the paper.

II. Preliminaries

A. Circuit Terminology

Our work focuses on multi-level combinational circuits, which can be modeled by a directed acyclic graph with nodes representing input pins and logic gates, and directed edges representing wires connecting the gates. The **primary inputs** (**PIs**) are source nodes of the graph. The **primary outputs** (**POs**) are a subset of the nodes of the graph. A widely-used circuit representation is **AND-Inverter Graph** (**AIG**), in which each node is either a PI or a two-input AND gate, and each edge optionally contains a marker indicating logical negation.

The inputs of a node are called its **fanins**. If there is a path from node A to B, A is a **transitive fanin** (**TFI**) of B. The **TFI cone** of a node includes the node itself and its TFIs [15].

The **don't-cares** for a logic function are those input patterns that allow its output to be either 0 or 1. All don't-care patterns constitute the **don't-care set** of the function, while the complement is the **care set**. Due to the complement relationship, we only focus on the care set in the rest of the paper.

An **irredundant sum-of-products** (**ISOP**) of a function is a sum-of-product expression, in which each product term is a prime implicant and no product term can be deleted without changing the function [16].

B. Error Metrics

Error metrics are essential for measuring the accuracy of approximate circuits. The metrics relevant to our work are **error rate** (**ER**) and **error distance** (**ED**).

In a circuit with I PIs and O POs, let the set of all possible input combinations be $\{\mathbf{x}_1, \dots, \mathbf{x}_{2^I}\}$. Assume that $\mathbf{x}_i$ $(1 \le i \le 2^I)$ occurs with a probability p_i. Let $\hat{\mathbf{y}}_i$ and $\mathbf{y}_i$ be the values encoded by the approximate and accurate output vectors for input vector $\mathbf{x}_i$, respectively.

ER is calculated as the probability that the outputs are incorrect:

$$ER = \sum_{1 \le i \le 2^I : \hat{\mathbf{y}}_i \ne \mathbf{y}_i} p_i.$$

It can evaluate the accuracy of random/control circuits, such as voters and decoders.

The difference between $\hat{\mathbf{y}}_i$ and $\mathbf{y}_i$ is known as the **error distance** (**ED**). In practice, two statistical measures on ED are typically used, which are **normalized mean error distance** (**NMED**) and **mean relative error distance** (**MRED**) [17]. They can measure the accuracy of arithmetic circuits, such as adders and multipliers.

NMED is the mean ED normalized by the maximum output value, defined as follows:

$$NMED = \sum_{i=1}^{2^I} \frac{|\hat{\mathbf{y}}_i - \mathbf{y}_i| \cdot p_i}{2^O - 1}.$$

MRED is the mean of the relative EDs, defined as follows:

$$MRED = \sum_{i=1}^{2^I} \frac{|\hat{\mathbf{y}}_i - \mathbf{y}_i| \cdot p_i}{\max\{\mathbf{y}_i, 1\}}.$$

To avoid division by zero, the denominator above is set as the maximum of the correct output value and 1.

III. Methodology

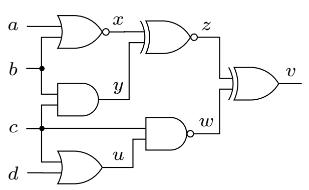
In this section, we present the technique to generate the approximate care set. Then, we introduce the proposed LAC, which is a resubstitution using this care set. Finally, we present an ALS flow based on the LAC.

A. Approximation of Care Set

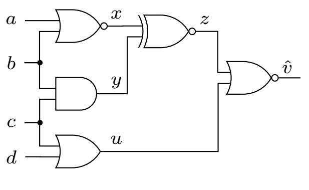
We generate the set of approximate care patterns for a node function using logic simulation with random input patterns following a user-specified distribution.

Assume that a node function is to be expressed using a set of nodes belonging to the same circuit, called **divisors**. We call the patterns appearing at the divisors after logic simulation the **approximate cares of the node at the divisors**. All PI patterns producing the approximate cares at the divisors are called the **approximate cares of the node at the PIs**. In what follows, unless otherwise specified, we use **approximate cares** to refer to approximate cares at the divisors. Since only some of patterns are selected as the approximate cares, such an approximation shrinks the care set. The following example shows how to generate an approximate care set.

Example 1. *The circuit shown in Fig. 1a has 4 inputs a, b, c, and d, with 16 PI combinations in total. The node values under all PI patterns are listed in Table I. To derive an approximate function of node v with divisors $\{u, z\}$, we randomly select 5 PI patterns $abcd = \{0000, 0010, 0011, 0100, 1000\}$ (see the shaded rows in Table I) and simulate the circuit. The patterns $\{00, 01, 10\}$ appearing at divisors $g = \{u, z\}$ are the approximate cares of node v at g. Among all the 16 PI combinations, 11 patterns produce the approximate cares at g. They are the approximate cares of node v at the PIs. Based on this, v's function can be simplified from the original expression $v = z \oplus w$ to $\hat{v} = \neg(u \vee z)$ (see Example 4 for details). The resulting circuit is shown in Fig. 1b, which is simplified compared to the original one. If the inputs are uniformly distributed, such a simplification introduces 18.75% error rate at node v (i.e., 3 in 16 PI patterns generate incorrect outputs).*



(a) Original circuit.



(b) Approximate circuit.

Fig. 1: Example circuits.

Our approximation of the care set is expressed by representing care patterns at the set of divisors. Compared to the previous approaches based on approximate EXDC [6], [7], our representation scales better for large circuits since EXDCs are expressed at the PIs. Indeed, in Example 1, the 3 approximate cares at divisors $\{u, z\}$ correspond to 11 approximate cares at the PIs. It implies that a few approximate cares expressed with divisors are equivalent to many approximate

TABLE I: Node values under all PI patterns for the circuit in Fig. 1a.

$abcd$	x	y	**u**	**z**	w	v	$abcd$	x	y	**u**	**z**	w	v
0000	1	0	**0**	**0**	1	1	1000	0	0	**0**	**1**	1	0
0001	1	0	**1**	**0**	1	1	1001	0	0	**1**	**1**	1	0
0010	1	0	**1**	**0**	0	0	1010	0	0	**1**	**1**	0	1
0011	1	0	**1**	**0**	0	0	1011	0	0	**1**	**1**	0	1
0100	0	0	**0**	**1**	1	0	1100	0	0	**0**	**1**	1	0
0101	0	0	**1**	**1**	1	0	1101	0	0	**1**	**1**	1	0
0110	0	1	**1**	**0**	0	0	1110	0	1	**1**	**0**	0	0
0111	0	1	**1**	**0**	0	0	1111	0	1	**1**	**0**	0	0

cares expressed with PIs. Therefore, even in large circuits with many PIs, we can store the approximate care patterns without a large memory overhead, while they have the same expressive power as the approximate cares at the PIs.

B. Proposed Local Approximate Change

In this section, we present a novel LAC by resubstituting a node function with another function derived using the approximate care set. The original node function is represented with its fanins, while the **approximate resubstitution function** is expressed with some divisors that are not necessarily its fanins. In Example 1, the original node function for v is $z \oplus w$. It is replaced by divisors $\{u, z\}$ with a resubstitution function $v = \neg(u \vee z)$, using the approximate care set $uz = \{00, 01, 10\}$.

To generate approximate resubstitutions for a node, three basic questions need to be answered:

- How to select good divisors among internal nodes in the circuit?
- Is a given set of divisors feasible to perform approximate resubstitution of the function?
- Given a feasible divisor set and an approximate care set, how to derive an approximate resubstitution function of the node, which can replace the original function?

We answer these three questions in Sections III-B1, III-B2, and III-B3, respectively. After that, we present how to generate LAC candidates in Section III-B4.

1) Selection of Divisors: For each node, there are numerous divisor sets producing different resubstitutions. We only select part of them for the sake of efficiency. Our strategy is illustrated in Algorithm 1. Specifically, the divisor sets for a node V are generated by the following operations.

- Remove a fanin n from the fanin set of node V (Lines 5–6).
- Replace a fanin n in the fanin set of node V with another node u in V's TFI cone (Lines 7–9).

Only the divisors in V's TFI cone are considered, since the function of V more likely depends on them, instead of the divisors outside the cone.

Algorithm 1: *Select_Divisor_Sets*(V, G)

```
   Input: node $V$, circuit $G$
   Output: divisor sets $S$
1  Find the TFI cone $T$ of node $V$ in circuit $G$;
2  Sort nodes in $T$ in ascending order of logic levels;
3  Divisor sets $S \leftarrow \emptyset$, node set $FI \leftarrow$ V's fanins;
4  for each node $n$ in $FI$ do
5      Divisor set $A \leftarrow FI \backslash \{n\}$; /* remove a fanin $n$ */
6      Add $A$ into $S$;
7      for each node $u$ in TFI cone $T$ do
8          Divisor set $B \leftarrow A \cup \{u\}$; /* replace a fanin $n$ by $u$ */
9          Add $B$ into $S$;
10 return $S$;
```

2) Feasibility of Divisors: Note that given a set of divisors, it may be impossible to derive a function with them to substitute the original node function. For instance, in Fig. 1a, we cannot find a function in terms of a and b to resubstitute v. The reason is that v not only depends on a and b, but also on c and d. Thus, for a divisor set generated from Algorithm 1, it is necessary to check the feasibility of the divisors to form a valid resubstitution function. Our proposed check method is based on the following theorem for checking the feasibility of an **accurate** resubstitution [18].

Theorem 1. *Assume that there are m divisors with functions $g_1(\mathbf{x})$, $g_2(\mathbf{x})$, …, $g_m(\mathbf{x})$ and a node V with the function $F(\mathbf{x})$, where $\mathbf{x}$ is the set of PI variables. The divisors can form an accurate resubstitution function of node V, if and only if there are no PI patterns $\mathbf{x}_1$ and $\mathbf{x}_2$, such that $F(\mathbf{x}_1) \neq F(\mathbf{x}_2)$, but $g_j(\mathbf{x}_1) = g_j(\mathbf{x}_2)$ for all $1 \leq j \leq m$.*

The theorem can be explained intuitively as follows. If there is a function $H(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$ that can accurately resubstitute $F(\mathbf{x})$, then for any PI patterns $\mathbf{x}_1$ and $\mathbf{x}_2$ satisfying $g_j(\mathbf{x}_1) = g_j(\mathbf{x}_2)$ for all j's, we must have $F(\mathbf{x}_1) = H(g_1(\mathbf{x}_1), \ldots, g_m(\mathbf{x}_1)) = H(g_1(\mathbf{x}_2), \ldots, g_m(\mathbf{x}_2)) = F(\mathbf{x}_2)$. A simple application of Theorem 1 is as follows.

Example 2. *For the circuit in Fig. 1a, to check whether divisors $\{u, z\}$ can accurately resubstitute v with a function, we enumerate all the 16 PI patterns. Under PI patterns $abcd = 0001$ and 0010, node v takes different values. Yet the patterns on $\{u, z\}$ are the same (i.e., $uz = 10$). By Theorem 1, it is impossible to derive a function in terms of $\{u, z\}$ to accurately resubstitute v.*

In [18], Theorem 1 is converted into a SAT instance to check the feasibility of divisors. It is time-consuming for large circuits. For approximate computing, it is not necessary to check all the PI combinations with a SAT solver. Instead, we only check PI patterns appearing in the logic simulation. If Theorem 1 is satisfied under PI patterns appearing in limited simulation rounds, then the given divisors can form an approximate resubstitution function.

Example 3. *Consider the circuit in Fig. 1a. Assume that the same PI patterns of Example 1, i.e., $abcd = \{0000, 0010, 0011, 0100, 1000\}$ (see the shaded rows in Table I), are selected to perform logic simulation. We check whether divisors $\{u, z\}$ can resubstitute v with an approximate function under these simulation patterns. In this case, the corresponding patterns on the divisor set are $uz = \{00, 10, 10, 01, 01\}$, while v takes the value $\{1, 0, 0, 0, 0\}$. We can see that each pattern on $\{u, z\}$ corresponds to only one value of v (i.e., 00, 01, and 10 correspond to 1, 0, 0, respectively). Thus, Theorem 1 holds for the set of random simulation patterns. Therefore, it is possible to derive an approximate function in terms of $\{u, z\}$ to resubstitute v. In other words, $\{u, z\}$ is a feasible divisor set.*

3) Derivation of Approximate Resubstitutions: At this point, we have selected feasible divisor sets. We also have the approximate cares on each divisor set. Approximate resubstitution functions will be derived from them. Instead of utilizing a SAT-based method in [14] or a BDD-based method in [18], we efficiently generate approximate resubstitution functions by computing ISOP expressions.

For node V and a feasible divisor set g of V, we build the truth table of an approximate resubstitution function for V on the approximate cares at g. Its input variables are the nodes in g and the output variable is node V. In the truth table, if an input combination is not in the approximate care set at nodes in g, its output value is a don't-care ("$-$"). Otherwise, the output value is the value of V for the corresponding approximate care pattern. Note that since the divisor set is feasible, although there may be many PI patterns producing the same approximate care pattern, the value of node V for all of them is the same.

Next, we compute an ISOP expression from the truth table as the approximate resubstitution function using Espresso [19]. After that, the ISOP expression will be converted to some nodes in the circuit. Then, the new nodes are used to replace the original node V and change the local structure of the circuit.

Example 4. *From Example 3, for the circuit in Fig. 1a, there exist approximate functions in terms of* $\{u, z\}$ *to resubstitute* v *if logic simulation is performed with 5 PI patterns* $abcd = \{0000, 0010, 0011, 0100, 1000\}$. *To derive an approximate function, a truth table shown in Table II is built with inputs* u *and* z *and output* $\hat{v}$. *The pattern* $uz = 11$ *is not in the approximate care set, so it is a don't care pattern. For the approximate care patterns* $uz = \{00, 01, 10\}$, *the corresponding output values are set as* $\hat{v} = \{1, 0, 0\}$, *which can be directly obtained from Table I. If* $\hat{v}$ *takes 0 when* $uz = 11$, *a possible ISOP expression is* $\hat{v} = \neg u \wedge \neg z$, *and it is converted into an NOR gate. Then, gates* w *and* v *in Fig. 1a are removed and NOR gate* $\hat{v}$ *is added into the circuit, producing the approximate circuit shown in Fig. 1b.*

TABLE II: A truth table of an approximate function with inputs u and z and output $\hat{v}$ in Example 4.

uz	00	01	10	11
$\hat{v}$	1	0	0	–

4) Generation of LAC Candidates: After showing the answers to the three basic questions, we present our method to generate LAC candidates. It is shown in Algorithm 2.

```
Algorithm 2: Generate_LACs(G, N, L)
  Input: circuit $G$, simulation round $N$, limit of LAC count $L$
  Output: LAC candidate set $\Pi$
1 Perform logic simulation for $N$ rounds;
2 LAC candidate set $\Pi \leftarrow \emptyset$;
3 for each node $V$ in $G$ do
4     Divisor sets $S \leftarrow$ Select_Divisor_Sets$(V, G)$;
5     LAC count $l \leftarrow 0$;
6     for each divisor set $g$ in $S$ do
7         if $l \geq L$ then break;
8         if $g$ is feasible to resubstitute $V$ then
9             $l \leftarrow l + 1$;
10            Build truth table $B$ with input set $g$ and output $V$;
11            New function $f \leftarrow$ Find_ISOP$(B)$;
12            Add function $f$ into candidate LAC set $\Pi$;
13 return $\Pi$;
```

The inputs of Algorithm 2 are a circuit G, a simulation round N, and the maximum number of LAC candidates L, while its output is a LAC candidate set Π. First, we perform N rounds of logic simulation in circuit G (Line 1). Then, for each node, we select some divisor sets S by Algorithm 1 (Line 4). After that, the feasibility of each divisor set (Line 8) is checked. If it is feasible, the truth table B with input set g and output V are generated (Line 10). Based on this, an approximate resubstitution function f is derived by computing an ISOP expression and added into the LAC candidate set Π (Lines 11–12). To limit the total number of LACs, at most L LACs are generated for each node V (Line 7).

C. Proposed Approximate Logic Synthesis Flow

In this section, we present ALSRAC, an ALS flow based on the proposed LACs, which is shown in Algorithm 3.

```
Algorithm 3: ALSRAC flow.
  Input: original circuit $G_{in}$, error threshold $E_t$, initial
         simulation round $N$, limit of LAC count $L$, controlling
         parameter $t$, scaling factor $r$ $(0 < r < 1)$.
  Output: approximate circuit $G_{out}$
1 Circuit $G \leftarrow$ Convert_To_AIG$(G_{in})$, $E \leftarrow 0$;
2 while $E \leq E_t$ do
3     Generate $N$ PI patterns randomly;
4     Candidate LAC set $\Pi \leftarrow$ Generate_LACs(G, N, L);
5     if $\Pi \neq \emptyset$ then
6         Find the best LAC in $\Pi$ with the smallest error $E$;
7         if $E > E_t$ then break;
8         Apply the best LAC with the smallest error to $G$;
9         Optimize $G$ with traditional logic synthesis tool;
10    if $\Pi = \emptyset$ for continuous $t$ iterations then $N \leftarrow N \times r$;
11 $G_{out} \leftarrow$ Technology_Mapping$(G)$;
12 return $G_{out}$
```

The inputs of the algorithm are an accurate circuit G_{in}, an error threshold E_t, an initial simulation round N, a limit of LAC count L, a controlling parameter t, and a scaling factor r. The controlling parameter and the scaling factor are related to the adjustment of N, which will be described later. The flow works on an AIG iteratively and returns an approximate design G_{out} meeting the error threshold. For each iteration, if the error is no more than the threshold, the LAC candidates are generated by Algorithm 2 (Line 4). Then, we evaluate the induced errors of LACs and find the best one with the smallest error E (Line 6). In order to evaluate the errors of all LACs efficiently, we apply the batch error estimation method [13], which has the same error estimation accuracy as applying traditional simulation to each LAC individually, but is much faster. If the error E does not exceed the error threshold E_t (Line 7), the best LAC is applied to simplify the circuit (Line 8). Due to the redundancies in the circuit after applying the LAC, traditional logic optimization is performed (Line 9). Finally, after the iteration loop terminates, we perform technology mapping and return the resulting approximate circuit satisfying the error constraint (Line 11).

One important feature of ALSRAC is that the simulation round N is adjusted dynamically to control the approximation degree. Theoretically, as N increases, the approximate care set approaches the accurate one. In this case, the LAC will be closer to the accurate function, and hence, the induced error of the LAC decreases. However, since a large N increases the size of approximate care set, it limits the approximation space. Therefore, sometimes finding a LAC from approximate care patterns with a large N is difficult. On the contrary, as N decreases, it is easier to find a LAC.

In our implementation, N is set to a suitable value initially. If the circuit to be approximated does not have any LAC candidates (i.e., $\Pi = \emptyset$), it means that the size of the care set is too large. In this case, we should shrink the approximate care set by reducing the simulation round N. Thus, more LAC candidates with larger induced error can be generated. However, different PI patterns generate distinct approximate care patterns. Therefore, when $\Pi = \emptyset$ for a set of PI patterns, we can try another set of PI patterns to get a different care set (Line 3), which may generate some LACs. Hence, N will not be reduced as soon as $\Pi = \emptyset$. Instead, we introduce a controlling parameter t to control the decreasing of simulation round N. N will be scaled by r $(0 < r < 1)$ only when $\Pi = \emptyset$ for t consecutive iterations (Line 10).

IV. Experimental Results

In this section, we present the experimental results of ALSRAC.

A. Experiment Setup

We implemented ALSRAC in C++ and tested it on a single core of AMD 3700X processor running at 3.8GHz with 16GB RAM. ER and ED are selected as the error metrics, which are measured by performing 10,000,000 rounds of logic simulation to guarantee good accuracy. All PI vectors are assumed to be uniformly distributed in our experiments, although our method is applicable to any PI distribution. The **area ratio** (the area of the approximate circuit over the original one) and the **delay ratio** (the delay of the approximate circuit over the original one) are used to evaluate the approximated designs. Both ASIC and FPGA designs are considered in our experiments. Particularly, the area and delay of an FPGA design are measured using its LUT count and the depth of its LUT network, respectively. It is obvious that smaller ratios are preferred due to more reduction in area and delay. The traditional optimization (Line 9 in Algorithm 3) is performed by ABC [20] using commands "*sweep; resyn2*".

Important parameters of our method are listed below. The number of simulation rounds N is initially set to 32. The limit on LAC counts at each node is $L = 1$. The simulation round N is scaled by $r = 0.9$ only if there are no LACs for successive $t = 5$ iterations.

It seems that the initial simulation round $N = 32$ is negligible compared to the number of all PI patterns, which possibly introduces large errors into the circuit. However, it does generate good approximate circuits in our experiments even for a small error threshold. The reason is that we work on an AIG (Line 1 in Algorithm 3). The care sets and the resubstitution functions are expressed with at most 2 divisors. Thus, even using just a few PI patterns, the approximate care set is still close to the accurate one.

In addition, due to the randomness of logic simulation, ALSRAC may produce different approximate circuits in different runs. The experiments have been performed three times and the average circuit quality and runtime are reported.

TABLE III: Benchmarks used in our experiments.

ISCAS & arithmetic			EPFL random/control			EPFL arithmetic		
Circuit	Area	Delay	Circuit	#LUT	Depth	Circuit	#LUT	Depth
alu4	2798	12.7	arbiter	409	23	adder	192	64
c1908	758	37.3	cavlc	101	6	shifter	512	4
c2670	1262	21.9	alu ctrl	27	2	divisor	3268	1208
c3540	1604	55.0	decoder	270	2	hyp	40406	4532
c5315	2451	47.5	i2c ctrl	227	7	log2	6574	119
c7552	2756	69.4	Int2float	28	6	max	523	189
c880	585	24.9	mem ctrl	2354	22	mult	4923	90
cla32	958	38.5	priority	110	26	sine	1229	55
ksa32	1128	17.8	router	52	6	sqrt	3077	1106
mtp8	1069	37.8	voter	1301	17	square	3246	74
rca32	666	16.1						
wal8	1081	45.3						

B. Experiments on ASIC Designs

We compare ALSRAC with a state-of-the-art ALS approach in [13], Su's method, on ASICs under ER and NMED constraints. Both methods aim at the best approximate circuits with the smallest area ratio satisfying the error constraint. Some ISCAS and arithmetic benchmarks are used. They are listed in the first column of Table III and are optimized with the logic synthesis tool SIS [21] before our experiments. They are exactly the same ones as those used in [13]. We reimplement Su's method with C++. The final approximate circuits in Su's method and ALSRAC are mapped with MCNC standard cell library [22] using the ABC command "*map -D <original delay>*".

1) Comparison under ER constraint: In this experiment, we compare ALSRAC and Su's method under ER constraint using the ISCAS and the arithmetic circuits. The area ratio, delay ratio, and runtime are listed in Table IV. The values for each benchmark are the average results under 7 ER thresholds (0.1%, 0.3%, 0.5%, 0.8%, 1%, 3%, 5%). The entries in **bold** highlight the cases where ALSRAC is better than Su's method. Similar notations are applied in the following tables. For all cases, ALSRAC reduces more area than Su's method. On average, ALSRAC generates approximate circuits with an area ratio of $\mathbf{80.11\%}$, improving over Su's method by $\mathbf{8.39\%}$ relatively. For *c7552*, *cla32*, *ksa32*, and *wal8*, our approach reduces more than $\mathbf{10\%}$ area over Su's method. Furthermore, all approximate designs other than *alu4* generated by ALSRAC have smaller delays compared to the accurate circuits. Nearly half of the approximate designs have better area and delay simultaneously than the circuits produced by Su's method. With regard to the runtime, ALSRAC is $\mathbf{80\times}$ faster than Su's method on average.

TABLE IV: Comparison of ALSRAC and Su's method under ER constraint.

Circuit	Average area ratio		Average delay ratio		Average time (s)	
	ALSRAC	Su's	ALSRAC	Su's	ALSRAC	Su's
alu4	**70.46%**	73.72%	101.35%	100.00%	**547**	10639
c1908	**75.24%**	77.33%	79.70%	76.98%	**19**	398
c2670	**66.75%**	74.60%	**89.89%**	97.46%	**6**	491
c3540	**92.89%**	94.66%	**86.47%**	100.00%	**199**	2063
c5315	**87.91%**	95.50%	**73.53%**	99.07%	**66**	3192
c7552	**80.30%**	91.13%	**77.97%**	94.96%	**153**	10275
c880	**90.48%**	93.58%	89.50%	87.49%	**20**	59
cla32	**59.72%**	79.69%	84.34%	58.66%	**6**	625
ksa32	**70.17%**	84.14%	91.17%	72.79%	**91**	1148
mtp8	**95.31%**	96.73%	**91.61%**	98.94%	**30**	548
rca32	**91.31%**	94.79%	99.56%	99.47%	**5**	28
wal8	**80.80%**	93.56%	95.90%	82.09%	**9**	1092
Arithmean	**80.11%**	87.45%	**88.42%**	88.99%	**32**	2546

2) Comparison under NMED constraint: In this experiment, we compare ALSRAC and Su's method under NMED constraint. Only the arithmetic circuits are selected, since NMED is an error metric for arithmetic circuits. Table V lists the area ratio, delay ratio, and runtime under the NMED constraint. The results for each benchmark are the averages under 8 NMED thresholds (0.00153%, 0.00305%, 0.00610%, 0.01221%, 0.02441%, 0.04883%, 0.09766%, 0.19531%). ALSRAC always reduces more area than Su's method. On average, our method produces approximate circuits with an area ratio of $\mathbf{39.64\%}$, improving over Su's method by $\mathbf{18.15\%}$ relatively. All approximate circuits generated by ALSRAC have smaller delays than the accurate ones. Two of them have better area and delay at the same time than the circuits generated by Su's method. Additionally, our method has a speed-up of $\mathbf{3\times}$ over Su's method on average.

TABLE V: Comparison of ALSRAC and Su's method under NMED constraint.

Circuit	Average area ratio		Average delay ratio		Average time (s)	
	ALSRAC	Su's	ALSRAC	Su's	ALSRAC	Su's
cla32	**15.85%**	26.03%	58.34%	52.73%	**404**	2723
ksa32	**16.11%**	26.16%	87.78%	78.86%	**714**	4498
mtp8	**78.60%**	82.65%	**97.52%**	97.62%	**2044**	2254
rca32	**23.48%**	28.68%	**86.57%**	95.81%	**514**	706
wal8	**64.14%**	78.63%	90.70%	89.74%	**758**	3592
Arithmean	**39.64%**	48.43%	84.18%	82.95%	**887**	2754

C. Experiments on FPGA Designs

We compare ALSRAC with a state-of-the-art ALS approach in [5], Liu's method, on FPGAs under ER and MRED constraints. Both methods aim at finding the best approximate circuit with the smallest area ratio satisfying the error constraint. The EPFL benchmarks are used to test the performance on FPGA designs. They are listed in the last two columns of Table III, which show the best synthesis results obtained in recent years. The final approximate circuits are mapped into 6-LUTs using the ABC command "*if -K 6*".

1) Comparison under ER constraint: In this experiment, we compare ALSRAC and Liu's method using EPFL random/control benchmarks using the ER threshold of 1%. The area ratio, delay ratio, and runtime are listed in Table VI. In all cases, ALSRAC reduces more area than Liu's method. On average, our method generates approximate circuits with an area ratio of $\mathbf{74.30}\%$, improving over Liu's method by $\mathbf{7.41}\%$. Meanwhile, the average delay ratio is improved by $\mathbf{10.60}\%$. In particular, only $\mathbf{9.09}\%$ of LUTs are needed to approximate the *priority* circuit. All approximate circuits generated by ALSRAC have delays no larger than the accurate ones. Half of them have both better area and delay compared to Liu's designs. Only the runtime of our approach is provided, since that of Liu's method is not mentioned in [5]. Our algorithm is efficient for all cases, even for the largest circuit *mem ctrl* with 2354 6-LUTs, ALSRAC can produce an approximate design in an acceptable time.

TABLE VI: Comparison of ALSRAC and Liu's method under ER constraint.

Circuit	Area ratio		Delay ratio		ALSRAC
	ALSRAC	Liu's	ALSRAC	Liu's	time (s)
arbiter	**53.06%**	61.37%	**43.48%**	56.52%	39
cavlc	**93.07%**	99.01%	83.33%	83.33%	69
alu ctrl	**96.30%**	100.00%	100.00%	100.00%	0.2
decoder	**97.78%**	100.00%	100.00%	100.00%	5
i2c ctrl	**78.41%**	90.31%	85.71%	85.71%	24
Int2float	**89.29%**	92.86%	**83.33%**	100.00%	4
mem ctrl	**70.56%**	88.62%	**36.36%**	68.18%	2059
priority	**9.09%**	10.91%	11.54%	11.54%	2
router	**57.69%**	59.62%	**16.67%**	33.33%	2
voter	**97.77%**	99.85%	**99.85%**	100.00%	967
Arithmean	**74.30%**	80.25%	**66.03%**	73.86%	317

2) Comparison under MRED constraint: In this experiment, we compare ALSRAC and Liu's method using EPFL arithmetic benchmarks under the MRED threshold of 0.19531%. Notably, Liu's work uses MRED instead of NMED as the measure on ED. Thus, this experiment also uses MRED as the error metric. The area ratio, delay ratio, and runtime are listed in Table VII. All the EPFL arithmetic circuits are listed in the table except *hyp*, since it has a massive number (40406) of 6-LUTs and ALSRAC cannot synthesize it within 24 hours. In terms of performance, ALSRAC reduces more area than Liu's method for all the benchmarks except *divisor* and *max*. Especially for the circuit *max*, ALSRAC is not competitive compared to Liu's method. With/without the *max* circuit, our method generates approximate designs with an area ratio of $\mathbf{59.69}\%/\mathbf{56.20}\%$, improving by $\mathbf{1.48}\%/\mathbf{11.86}\%$ compared to Liu's method. In particular, for *shifter* and *mult*, ALSRAC reduces nearly $\mathbf{20}\%$ more area. Besides, only $\mathbf{3.09}\%$ of 6-LUTs are required to approximate the *sqrt* circuit. All approximate circuits generated by ALSRAC have delay no larger than the accurate ones. More than half of them have both area and delay not worse than Liu's method. The runtime is provided only for our approach because the runtime is not reported in [5]. Overall, the runtime of our method is acceptable.

V. Conclusion

This paper proposes ALSRAC, an approximate logic synthesis flow that relies on resubstitution with an approximate care set to produce high-quality approximate circuits. The main idea is to control the size of the approximate care set by logic simulation and find approximate changes that can be applied to the design. The experimental results show that ALSRAC significantly improves the quality of approximate logic synthesis.

TABLE VII: Comparison of ALSRAC and Liu's method under MRED constraint.

Circuit	Area ratio		Delay ratio		ALSRAC
	ALSRAC	Liu's	ALSRAC	Liu's	time (s)
adder	**68.23%**	71.35%	**4.69%**	15.63%	891
shifter	**71.48%**	90.04%	100.00%	100.00%	435
divisor	99.72%	98.90%	**70.94%**	88.41%	1982
log2	**90.07%**	97.37%	90.76%	90.76%	614
max	87.57%	35.18%	100.00%	10.05%	134
mult	**8.00%**	27.16%	**21.11%**	40.00%	25086
sine	**96.75%**	99.19%	**83.64%**	98.18%	52
sqrt	**3.09%**	10.98%	**1.81%**	10.13%	4238
square	**12.26%**	15.10%	27.03%	25.68%	9998
Arithmean	**59.69%**	60.59%	55.55%	53.20%	4826
Arithmean w/o max	**56.20%**	63.76%	**50.00%**	58.60%	5412

References

[1] J. Han and M. Orshansky, "Approximate computing: an emerging paradigm for energy-efficient design," in *ETS*, 2013, pp. 1–6.

[2] Y. Kim *et al.*, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *ICCAD*, 2013, pp. 130–137.

[3] C. Liu *et al.*, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *DATE*, 2014, pp. 95:1–95:4.

[4] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE TEVC*, vol. 19, no. 3, pp. 432–444, 2015.

[5] G. Liu and Z. Zhang, "Statistically certified approximate logic synthesis," in *ICCAD*, 2017, pp. 344–351.

[6] S. Venkataramani *et al.*, "SALSA: systematic logic synthesis of approximate circuits," in *DAC*, 2012, pp. 796–801.

[7] J. Miao *et al.*, "Multi-level approximate logic synthesis under general error constraints," in *ICCAD*, 2014, pp. 504–510.

[8] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *DAC*, 2016, pp. 128:1–128:6.

[9] Y. Yao *et al.*, "Approximate disjoint bi-decomposition and its application to approximate logic synthesis," in *ICCD*, 2017, pp. 517–524.

[10] D. Shin and S. Gupta, "A new circuit simplification method for error tolerant applications," in *DATE*, 2011, pp. 1–6.

[11] A. Chandrasekharan *et al.*, "Approximation-aware rewriting of AIGs for error tolerant applications," in *ICCAD*, 2016, p. 83.

[12] S. Venkataramani *et al.*, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *DATE*, 2013, pp. 1367–1372.

[13] S. Su *et al.*, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *DAC*, 2018, pp. 54:1–54:6.

[14] A. Mishchenko *et al.*, "Scalable don't-care-based logic optimization and resynthesis," *ACM TRETS*, vol. 4, no. 4, pp. 34:1–34:23, 2011.

[15] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, 2005, pp. 412–417.

[16] T. Sasao and J. Butler, "Worst and best irredundant sum-of-products expressions," *IEEE TC*, vol. 50, no. 9, pp. 935–948, 2001.

[17] H. Jiang *et al.*, "A review, classification, and comparative evaluation of approximate arithmetic circuits," *ACM JETC*, vol. 13, no. 4, pp. 60:1–60:34, 2017.

[18] A. Mishchenko *et al.*, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE TCAD*, vol. 25, no. 5, pp. 743–755, 2006.

[19] P. Mcgeer *et al.*, "ESPRESSO-SIGNATURE: a new exact minimizer for logic functions," *IEEE TVLSI*, vol. 1, no. 4, pp. 432–440, 1993.

[20] A. Mishchenko *et al.*, "ABC: a system for sequential synthesis and verification, release 90703," http://people.eecs.berkeley.edu/~alanmi/abc/, accessed July 3, 2019.

[21] E. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," University of California, Berkeley, Tech. Rep., 1992.

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