Efficient Resubstitution-Based Approximate Logic Synthesis

Chang Meng, *Member, IEEE*, Alan Mishchenko, *Senior Member, IEEE*, Weikang Qian, *Senior Member, IEEE*, and Giovanni De Micheli, *Life Fellow, IEEE*

***Abstract*—Approximate computing is an emerging paradigm for designing error-resilient applications. It reduces circuit area, power, and delay at the cost of introducing errors. This paper proposes a powerful technique, termed Approximate Resubstitution (AppResub), to approximately simplify the circuit. AppResub replaces a node's function with a simpler approximate function on existing nodes in the circuit to reduce the hardware cost. Leveraging AppResub, an efficient flow for approximate logic synthesis (ALS) is developed by iteratively applying a set of promising AppResubs for circuit simplification. To evaluate errors caused by a set of AppResubs, a novel error model capable of efficiently computing an error upper bound is used to smartly apply AppResubs in the ALS flow. The experimental results demonstrate that compared to a state-of-the-art method, the proposed flow further reduces 20.9% area and 21.7% delay under the mean error distance constraint, while being 400× faster. The code of our flow is open-source.**

***Index Terms*—approximate logic synthesis, approximate computing, resubstitution**

I. Introduction

As the power consumption of digital systems grows rapidly, energy efficiency emerges as a pivotal concern [1]. Many prevalent applications, including image processing, data mining, and machine learning, inherently tolerate some degree of error, paving the way for a design paradigm called *approximate computing*. This paradigm modifies functions of computing systems by deliberately introducing some errors. If errors are carefully introduced, the application-level quality is almost unaffected, while the area, delay, and power of the system can be reduced dramatically.

Approximate computing can be applied to various layers of computing systems [2], including the circuit, architecture, and software layers. This work focuses on approximate computing at the circuit layer, aiming to design high-quality approximate circuits that balance low hardware costs against an acceptable error margin. The strategies to obtain such circuits fall into two main categories: *manual design* and *approximate logic synthesis* (*ALS*) [3]. Manual design predominantly targets arithmetic circuits, like adders [4]–[7] and multipliers [8]–[10]. Since arithmetic circuits have well-known regular structures, they are amenable to manual approximation. ALS, however, is not restricted to arithmetic circuits and is applicable to a wider range of circuits. It takes as inputs an accurate circuit and specific error constraints (*e.g.*, maintaining an error rate below 1%). Then, it automatically generates an approximate circuit satisfying the constraints with minimized hardware costs, including area, delay, and power. Our work studies ALS applied to general combinational circuits, targeting minimizing circuit area under a given error upper bound.

Most existing ALS methods simplify circuits by making local structural modifications, known as *local approximate changes* (*LACs*). However, existing LACs have drawbacks in two aspects. On the one hand, some LACs may introduce large errors. For example, a common yet simple LAC replaces a circuit node by a constant zero or one [11]–[16]. While straightforward and widely used, they tend to introduce large errors. On the other hand, the generation of certain LACs requires a substantial amount of computation. For instance, a state-of-the-art LAC is based on Boolean matrix factorization [17], [18], which approximately decomposes a sub-circuit into a compressor unit and a decompressor unit. This decomposition effectively simplifies the circuit. However, Boolean matrix factorization is a complex process demanding substantial computational effort, making it a time-consuming approach to achieve approximation.

To address the above challenges, we introduce a novel LAC called *approximate resubstitution* (*AppResub*). AppResub simplifies a node by approximately re-expressing its function using a set of other nodes in the circuit. Utilizing multiple nodes to express a new function makes AppResub more expressive than traditional LACs, thereby usually resulting in smaller errors. Moreover, we generate AppResubs with logic simulation, which is efficient and scalable for large circuits.

Our main contributions are as follows:

1) We propose a powerful LAC called AppResub, which has stronger expressive ability and induces smaller errors, compared to the traditional LACs. For efficient AppResubs generation, we devise an approach based on logic simulation. It quickly identifies AppResub opportunities by analyzing simulation patterns, treats unobserved patterns as *don't cares*, and constructs truth tables of approximate functions for resubstitution.
2) We present an error model to estimate the error upper bound caused by a set of AppResubs. It has low computational complexity and assists in selecting a promising set of AppResubs. Using the model usually ensures that after applying the selected AppResubs, the resulting approximate circuit satisfies the specified error constraint.

Chang Meng is with the Integrated Systems Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland (email: changmeng@epfl.ch).

Alan Mishchenko is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA (email: alanmi@berkeley.edu).

Weikang Qian is with the University of Michigan-Shanghai Jiao Tong University Joint Institute and the MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, China (email: qianwk@sjtu.edu.cn).

Giovanni De Micheli is with the Integrated Systems Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland (email: giovanni.demicheli@epfl.ch).

Corresponding author: Weikang Qian.

3) We develop an efficient resubstitution-based ALS flow. It works by iteratively applying a set of promising AppResubs to the circuit. To select promising AppResubs, we apply the above error model to convert the selection process into a knapsack problem and propose an efficient solution by solving its dual problem.

Our flow is applicable to any average error metric, including error rate, mean error distance, mean Hamming distance, and mean square error. The experimental results reveal substantial improvements in the quality of approximate circuits across various benchmarks and error metrics. For instance, compared to a state-of-the-art method under the mean error distance constraint, our flow achieves additional savings of 20.9% in area and 21.7% in delay. Our source code is available at https://github.com/changmg/ResubALS.

This paper expands a preliminary version previously published in [19]. The additional technical contributions lie in Contributions 2) and 3). Specifically, we propose a novel model on error upper bound, and based on it, we build an efficient ALS flow. Moreover, we conduct extensive experiments to demonstrate the scalability and broad applicability of our method.

The remainder of this paper is organized as follows. Section II introduces the background. Section III reviews the related works. Section IV elaborates the proposed LAC, *i.e.*, AppResub. Section V presents the resubstitution-based ALS flow. The experimental results are presented in Section VI, followed by conclusions in Section VII.

II. Background

A. Logic Circuit Terminologies

Our study focuses on multi-level combinational logic circuits, which can be modeled as directed acyclic graphs. For simplicity, we use the term *circuit* to refer to a multi-level combinational logic circuit.

In a circuit, the inputs and outputs of a node are called its *(direct) fanins and fanouts*, respectively. A *primary input* (*PI*) is a node without any fanin. A *functional node* is one performing a logic operation. A *primary output* (*PO*) is a dummy node driven by either a functional node or a PI; it has a single fanin and no fanouts. A *path* is a series of connected nodes in the circuit. If there exists a path from node u to v, then u is a *transitive fanin* (*TFI*) of v, and v is a *transitive fanout* (*TFO*) of u.

An *AND-inverter graph* (*AIG*) is a specific type of circuit, where each functional node is a two-input AND gate. Edges in an AIG can be complemented or non-complemented, with a complemented edge denoting a signal negation. Fig. 1 shows an AIG with 4 PIs a, b, c, d, 7 two-input AND gates r, s, t, u, v, w, y, and 1 PO y_o. Here, dashed lines represent complemented edges, while solid lines stand for non-complemented edges. For example, node v receives a complemented edge from node r and a non-complemented edge from node s, yielding the function $v = \bar{r}s$ (where $\bar{r}$ denotes the negation of r).

B. Error Metrics

Error metrics are used to evaluate the accuracy of approximate circuits. This work focuses on an important class of error metrics called *average errors*. Consider two

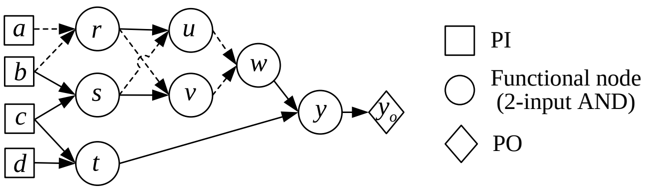


Fig. 1: An example AIG. Each functional node represents a two-input AND gate. The dashed lines indicate complemented edges, and the solid lines indicate non-complemented edges.

multiple-output Boolean functions $\boldsymbol{y} : \mathbb{B}^I \to \mathbb{B}^O$ for an accurate circuit $\mathbb{C}_{acc}$ and $\hat{\boldsymbol{y}} : \mathbb{B}^I \to \mathbb{B}^O$ for its approximate counterpart $\mathbb{C}$. Denote the numbers of PIs and POs of the circuits by I and O, respectively. The average error of circuit $\mathbb{C}$, represented as *Error*($\mathbb{C}$), quantifies the average deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ over all PI patterns:

$$\mathit{Error}(\mathbb{C}) = \sum_{\boldsymbol{x} \in \mathbb{B}^I} D(\boldsymbol{y}(\boldsymbol{x}), \hat{\boldsymbol{y}}(\boldsymbol{x})) \cdot p(\boldsymbol{x}), \tag{1}$$

where $\boldsymbol{y}(\boldsymbol{x})$ and $\hat{\boldsymbol{y}}(\boldsymbol{x})$ are binary vectors of length O, denoting the PO values of the circuits $\mathbb{C}_{acc}$ and $\mathbb{C}$ under the PI pattern $\boldsymbol{x}$, respectively, $p(\boldsymbol{x})$ is the occurrence probability of the PI pattern $\boldsymbol{x}$, and D represents a deviation function that quantifies the deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$.

Typical average errors include error rate, mean error distance, mean Hamming distance, and mean square error. Error rate is the probability of a PI pattern yielding an incorrect output in the approximate circuit. Its deviation function is defined as:

$$D_{\text{ER}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \begin{cases} 0 & \textit{if } \boldsymbol{y} = \hat{\boldsymbol{y}}, \\ 1 & \textit{if } \boldsymbol{y} \neq \hat{\boldsymbol{y}}. \end{cases} \tag{2}$$

Mean error distance measures the average absolute difference between the numerical values encoded by the POs of the accurate and approximate circuits. Its deviation function is given by:

$$D_{\text{MED}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \left| \mathit{int}(\boldsymbol{y}) - \mathit{int}(\hat{\boldsymbol{y}}) \right|, \tag{3}$$

where $\mathit{int}(\boldsymbol{v})$ returns the integer encoded by the binary vector $\boldsymbol{v}$. For example, if $\boldsymbol{y}$ encodes an O-bit unsigned integer, then $\mathit{int}(\boldsymbol{y}) = \sum_{k=1}^{O} 2^{k-1} y_k$, where y_k denotes the k-th bit of the binary vector $\boldsymbol{y}$.

Besides, mean Hamming distance is the average count of bit-flips in $\hat{\boldsymbol{y}}$ compared to $\boldsymbol{y}$. Its deviation function is $D_{\text{MHD}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{k=1}^{O} |y_k - \hat{y}_k|$. Mean square error measures the average of the squares of errors between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. Its deviation function is $D_{\text{MSE}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = [\mathit{int}(\boldsymbol{y}) - \mathit{int}(\hat{\boldsymbol{y}})]^2$.

In practice, the average error of the approximate circuit $\mathbb{C}$ is commonly evaluated with Monte Carlo simulation by sampling M PI patterns $\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^M$, as shown below:

$$\mathit{Error}(\mathbb{C}) = \frac{1}{M} \sum_{i=1}^{M} D(\boldsymbol{y}(\boldsymbol{x}^i), \hat{\boldsymbol{y}}(\boldsymbol{x}^i)). \tag{4}$$

Additionally, the *normalized mean error distance* and *normalized mean Hamming distance* are defined as follows:

$$\text{norm.\_mean\_error\_distance} = \frac{\text{mean\_error\_distance}}{2^O - 1},$$

$$\text{norm.\_mean\_Hamming\_dist.} = \frac{\text{mean\_Hamming\_dist.}}{O}.$$

C. Approximation Miter

An *approximation miter* is an auxiliary circuit used to evaluate errors [20]. Illustrated in Fig. 2, the miter implements the deviation function $D(\boldsymbol{y}, \hat{\boldsymbol{y}})$ in Eq. (1). It

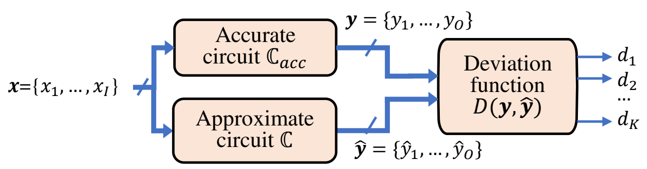

Accurate circuit $\mathbb{C}_{acc}$
Approximate circuit $\mathbb{C}$
$\boldsymbol{x}=\{x_1,\ldots,x_I\}$
$\boldsymbol{y}=\{y_1,\ldots,y_O\}$
$\hat{\boldsymbol{y}}=\{\hat{y}_1,\ldots,\hat{y}_O\}$
Deviation function $D(\boldsymbol{y},\hat{\boldsymbol{y}})$
d_1
d_2
d_K

Fig. 2: Approximation miter used for error evaluation [20].

takes the PIs $\boldsymbol{x}$ from both the accurate and approximate circuits as inputs. It has $K \geq 1$ outputs, *i.e.*, $d_1, d_2, \ldots, d_K$, capturing the computed deviation function D. For instance, an approximation miter for error rate has a single output d_1, representing the deviation D_{ER} defined in Eq. (2). On the other hand, a miter for mean error distance has $K = O$ outputs, encoding the deviation D_{MED} in Eq. (3).

Using the approximation miter, Eq. (4) can be reformulated as:

$$Error(\mathbb{C}) = \frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{K} 2^{k-1} d_k(\boldsymbol{x}^i), \quad (5)$$

where $d_k(\boldsymbol{x}^i)$ is the value of the k-th output of the approximation miter under the PI pattern $\boldsymbol{x}^i$.

III. Related Works

In this section, we review existing works closely related to our study. Since we propose an LAC called AppResub, we introduce existing LACs and highlight their relevance to AppResub in Section III-A. Moreover, because we propose a novel error upper bound model for ALS, we also review existing error estimation techniques in ALS.

A. Existing Local Approximate Changes (LACs) in ALS

Most ALS methods introduce approximation into circuits by applying LACs. Among them, the simplest one is the constant LAC. It substitutes a circuit node with a constant 0 or 1, effectively reducing the circuit area and potentially the delay at the cost of introducing errors. Despite its simplicity, the constant LAC is widely used in many existing ALS methods. Shin and Gupta are the first to propose the constant LAC [11]. They applied it within gate netlists by substituting gates with constants. Schlachter *et al.* also employed constant LACs on gate netlists [12]. They proposed criteria to decide which gates to be substituted by constants based on the gates' significance and activity. Chandrasekharan *et al.* explored constant substitution in AIGs [13]. To simplify a node in an AIG, they selected a cut of the node and rewrote the cut with a constant 0. Scarabottolo *et al.* identified and removed the largest sub-circuit replaceable by constants without violating the error constraint [14]. Witschen *et al.* modeled the selection of constant LACs through cutpoints, converting ALS into a minimal unsatisfiable subset problem, aiming to apply the maximum number of constant LACs to minimize the circuit area [15]. Zhou *et al.* [21] and Lee *et al.* [16] utilized constant LACs to reduce the delay of approximate circuits and developed delay-driven ALS flows. In fact, the constant LAC is a special case of our proposed LAC, AppResub, which will be discussed in Section IV-B1.

Beyond constant LACs, there are also many other finer LACs. Venkataramani *et al.* proposed a LAC called SASIMI, which substitutes a node u with another node v or v's negation [22]. If the function of u is similar to that of v or v's negation, such a substitution induces a small error. After the substitution, the *maximum fanout-free cone* (*MFFC*) of node u can be removed, thereby reducing the area and possibly the delay. SASIMI can be seen as a special case of AppResub. Wu and Qian proposed a LAC called ANS, which deletes some literals from the Boolean expression of a node in the circuit [20], [23]. It can also be viewed as a special case of AppResub. Liu *et al.* proposed a stochastic ALS flow including various LACs, *i.e.*, substituting a gate with a constant, flipping a gate's output, and adding a gate [24]. The constant substitution of a gate is a special case of AppResub. Tam *et al.* [25] proposed an ALS flow under the error rate constraint. It applies two types of LACs, *i.e.*, constant LAC and SASIMI, and hence both LACs can be seen as a special case of AppResub. Ma *et al.* [18] proposed an ALS flow called BLASYS. Its LAC is based on Boolean matrix factorization, which approximately decomposes a sub-circuit into two units, *i.e.*, a compressor unit and a decompressor unit. The area and delay of the sub-circuit can be reduced after decomposition. While the BLASYS LAC is not a special case of AppResub, generating AppResub is more efficient than generating the BLASYS LAC. Meng *et al.* [26] proposed an efficient ALS flow under the maximum error constraint. They applied the constant LAC and the SASIMI LAC, and as discussed above, both LACs are special cases of AppResub. Rezaalipour *et al.* [27] proposed a novel ALS method called XPAT. Its LAC targets approximating a multiple-input and multiple-output sub-circuit, and encodes potential approximate sub-circuits into a parametrizable template. Then, a satisfiability modulo theories solver is called to identify approximate sub-circuits meeting the error constraint. Since XPAT's LAC approximates a sub-circuit with multiple outputs, it is not a special case of AppResub, which approximates a sub-circuit with a single output.

B. Existing Error Estimation Techniques in ALS

An essential component of ALS methods is estimating the errors introduced by LACs. Researchers have developed many methods to estimate the error caused by a single LAC. To estimate the average error metrics, such as *error rate* and *mean error distance*, Su *et al.* introduced the concept of *change propagation matrix* (*CPM*) [28], [29]. The CPM evaluates whether a value change at an internal node affects the circuit outputs or not. Utilizing the CPM alongside the specific value change at the internal node, the error caused by each LAC can be efficiently computed. To estimate the maximum error distance caused by a single LAC, Scarabottolo *et al.* proposed to partition a circuit into sub-circuits and then analyze the error propagation of these sub-circuits [30]. Furthermore, to estimate the bit error rate caused by a single LAC, Echavarria *et al.* proposed an error transition model that propagates the bit error rates through cascaded sub-circuits [31].

Beyond single LAC analysis, researchers have also developed several approaches to estimate the cumulative error caused by applying multiple LACs simultaneously. Our work introduces an error upper bound model that falls into this category. Wu and Qian proposed a linear model, estimating the overall error rate caused by multiple LACs as the sum of the local error rates calculated for each LAC applied in isolation [23]. This linear model is further applied in subsequent works, such as [32] and [33], to estimate other average error metrics like mean error distance.

IV. Proposed LAC: Approximate Resubstitution

In this section, we propose a LAC called *AppResub* to approximately simplify a circuit. We first introduce accurate resubstitution for traditional logic synthesis. Then, we introduce AppResub for ALS.

A. Accurate Resubstitution

Accurate resubstitution [34]–[36] is a powerful circuit simplification technique in traditional logic synthesis. It re-expresses a node's function using a set of nodes already existing in the circuit, while preserving the circuit functionality. These nodes used for the re-expression are referred to as *divisors*. An example of accurate resubstitution is provided below.

Example 1 *Consider the AIG shown in Fig. 3(a). We can accurately resubstitute node w using divisors $\{r, s\}$ with the function $w' = \bar{r}\bar{s}$. By doing so, the nodes u and v are removed, and the resulting AIG is shown in Fig. 3(b).*

This resubstitution maintains w's function, thus preserving the overall functionality of the circuit. A simple derivation is as follows: Given that $w = \bar{u}\bar{v} = \overline{u+v}$, $u = r\bar{s}$, and $v = \bar{r}s$, it follows that $w = \overline{r\bar{s} + \bar{r}s} = rs + \bar{r}\bar{s}$. Substituting $r = \bar{a}\bar{b}$ and $s = bc$, we have $w = \bar{a}\bar{b}bc + \bar{r}\bar{s} = \bar{r}\bar{s}$, which is exactly the function w'.

Given a node and a selected set of divisors, an important problem is to check the feasibility of achieving an accurate resubstitution for the node with these divisors. This has been addressed by a theorem presented in [37], which we describe as follows:

Theorem 1 *Consider a set of PIs denoted as $\boldsymbol{x}$. Assume that there exist m divisors with respective functions $g_1(\boldsymbol{x})$, $g_2(\boldsymbol{x})$, …, $g_m(\boldsymbol{x})$, and a target node with function $f(\boldsymbol{x})$. These divisors can form an accurate resubstitution function for node f,* ***if and only if*** *there are no two PI patterns $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ that satisfy:*

1) *$g_j(\boldsymbol{x}_1) = g_j(\boldsymbol{x}_2)$ for each $1 \le j \le m$, and*
2) *$f(\boldsymbol{x}_1) \neq f(\boldsymbol{x}_2)$.*

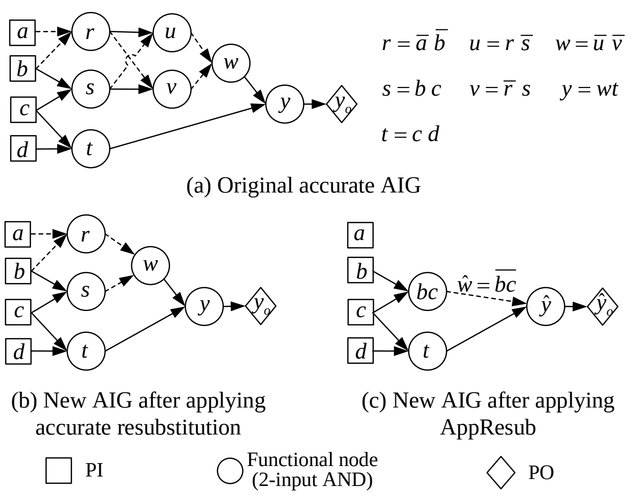
The essence of this theorem can be explained as follows. If there is a function $h(g_1(\boldsymbol{x}), \ldots, g_m(\boldsymbol{x}))$ that can accurately resubstitute $f(\boldsymbol{x})$, then for any PI patterns $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ making $g_j(\boldsymbol{x}_1) = g_j(\boldsymbol{x}_2)$ for each $1 \le j \le m$, we must have $f(\boldsymbol{x}_1) = h(g_1(\boldsymbol{x}_1), \ldots, g_m(\boldsymbol{x}_1)) = h(g_1(\boldsymbol{x}_2), \ldots, g_m(\boldsymbol{x}_2)) = f(\boldsymbol{x}_2)$. An example applying Theorem 1 is as follows.

Example 2 *Consider the AIG depicted in Fig. 3(a). To determine whether the divisors $\{r, s\}$ can be used to accurately resubstitute node w, we enumerate all 16 PI patterns for variables $\{a, b, c, d\}$ and simulate the AIG. The simulation results are listed in Table I. Note that under all PI patterns,*

- *when $rs = 00$, w is always 1*;
- *when $rs = 01$ or $rs = 10$, w is always 0*; *and*
- *$rs = 11$ never occurs.*

Therefore, no PI patterns $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ yield the same values for $\{r, s\}$ (satisfying condition 1), while producing different values for w (satisfying condition 2). According to Theorem 1, the divisors $\{r, s\}$ can be used to accurately resubstitute node w.

Conversely, $\{b, c\}$ fail to serve as a valid divisor set for an accurate resubstitution of node w. This is exemplified by the PI patterns $abcd = 0000$ and 1000, where the divisors $\{b, c\}$ yield the same pattern ($bc = 00$), but w takes different values 0 and 1, respectively. In this case, Theorem 1 does not hold.



(a) Original accurate AIG

(b) New AIG after applying accurate resubstitution

(c) New AIG after applying AppResub

Fig. 3: Example of accurate resubstitution and AppResub.

TABLE I: Simulation values under all PI patterns for the AIG in Fig. 3(a). This table is used in Examples 2–5. The shaded patterns are only used in Examples 4 and 5, denoting the randomly sampled patterns for generating an AppResub.

$abcd$	r	s	w	$\hat{w} = \overline{bc}$	$abcd$	r	s	w	$\hat{w} = \overline{bc}$
0000	1	0	0	1	1000	0	0	1	1
0001	1	0	0	1	1001	0	0	1	1
0010	1	0	0	1	1010	0	0	1	1
0011	1	0	0	1	1011	0	0	1	1
0100	0	0	1	1	1100	0	0	1	1
0101	0	0	1	1	1101	0	0	1	1
0110	0	1	0	0	1110	0	1	0	0
0111	0	1	0	0	1111	0	1	0	0

B. Approximate Resubstitution

For approximate computing, we propose *approximate resubstitution* (*AppResub*). It simplifies the circuit by *approximately* re-expressing a node n's function with a set of divisors, while the circuit functionality is not necessarily preserved. An example of AppResub is as follows.

Example 3 *From Example 2, we cannot resubstitute node w using the divisors $\{b, c\}$. However, if some errors are allowed, we can approximately resubstitute w using $\{b, c\}$ with the function $\hat{w} = \overline{bc}$. As shown in Table I, this AppResub causes errors on node w under 4 PI patterns: $abcd = 0000, 0001, 0010,$ and 0011. With this AppResub, nodes r, s, u, v, and w are removed. The AIG after the AppResub is shown in Fig. 3(c).*

For each node n in a circuit, there are numerous potential AppResubs, which involve different sets of divisors and errors. It is impractical to enumerate all AppResubs. Instead, we propose an efficient method to generate some *candidate AppResubs* for each node n. This process involves addressing the following three pivotal questions:

1) How to select appropriate divisors for re-expressing node n's function?
2) Given a divisor set $\boldsymbol{g}$, is there a function on $\boldsymbol{g}$ that can approximate n's function?
3) If the answer to Question 2) is yes, how to derive the corresponding function?

They will be answered in Sections IV-B1, IV-B2, and IV-B3, respectively. Based on the answers, we present an algorithm to efficiently generate candidate AppResubs in Section IV-B4.

1) Selecting Divisors: It is crucial to identify suitable divisors for each node n in the circuit, as not all nodes are suitable choices to re-express n's function. For instance, n's TFOs are unsuitable, since using a TFO as a divisor would create a dependency loop, which is not allowed in a combinational circuit. Our strategy only selects divisors from n's *divisor pool*. A node d is in n's divisor pool only if

- d is a TFI of n; or
- d is a direct fanout of node n's TFI, and the logic level of d is lower than that of n.

This choice ensures that divisors are likely to influence n's function. The logic level limitation avoids the case where d is n's TFO. This selection strategy is similar to that used in [36], [37] for accurate resubstitution.

After building n's divisor pool, we select m divisors from the pool to create a *divisor set* of n. We limit m to 0, 1, or 2 to manage complexity. Our experimental results suggest that this range yields high-quality approximate circuits with acceptable runtime.

Specifically, when $m = 0$, the set is empty. In this case, n is resubstituted by a constant, and AppResub degrades to a constant LAC, which is widely applied in [11]–[16], [21], [24]. When $m = 1$, AppResub utilizes a single divisor to resubstitute n, which is exactly the SASIMI LAC used in [22], [25], [26]. When $m = 2$, AppResub uses two divisors to resubstitute n. Note that the ANS LAC proposed in [20], [23] deletes some literals from the Boolean expression of a node in the circuit. If an ANS LAC is applied to a Boolean expression and eventually keeps two variables in the expression, then it is a special case of the AppResub, where the 2 divisors are exactly the two kept variables. For example, assume that an ANS LAC is applied to node w in the AIG shown in Fig. 3(b). Before applying the LAC, w's function is $w = \overline{r}\overline{s} = \overline{\overline{a}\overline{b}} \cdot \overline{bc}$, as we derived in Example 1. If an ANS LAC deletes the literals $\overline{a}$ and $\overline{b}$, then we have an approximate function $\hat{w} = \overline{bc}$, with the new AIG shown in Fig. 3(c). The ANS LAC in this example is exactly an AppResub on node w with 2 divisors b and c.

2) Checking Existence of AppResub: Given a divisor set for a node n, we can use Theorem 1 to check whether the divisor set can be used to accurately resubstitute n. Theorem 1 should be checked for all PI patterns, which is typically done by time-consuming SAT-based methods. However, for approximate computing, it is unnecessary to enumerate all PI patterns. We propose to check the conditions of Theorem 1 under some PI patterns encountered in random logic simulation. If Theorem 1 is satisfied under these PI patterns appearing in limited simulation rounds, then the given divisor set is considered as *a feasible divisor set*, which can be used to approximately resubstitute node n. Otherwise, it is infeasible and thus discarded.

Example 4 *Consider the AIG in Fig. 3(a). Assume that logic simulation randomly samples 5 PI patterns* $abcd =$ 0100, 0101, 0111, 1011, *and* 1110 *(refer to the shaded entries in Table I). Now, we check whether the divisors* $\{b, c\}$ *can approximately resubstitute node* w *under these PI patterns. In the simulation, the patterns on* $\{b, c\}$ *are* $bc = 10, 10, 11, 01$, *and* 11, *and the corresponding* w *values are* $1, 1, 0, 1$, *and* 0, *respectively. It is evident that each pattern on* $\{b, c\}$ *maps to a unique value of* w. *That is,* 10, 11, *and* 01 *map to* 1, 0, 1, *respectively. Thus, Theorem 1 holds under the 5 PI patterns, and* $\{b, c\}$ *is a feasible divisor set that can approximately resubstitute node* w.

3) Deriving AppResub: Given a feasible divisor set for a node n, we need to derive a new function h on the divisor set to approximately resubstitute n's function. Specifically, we employ the same PI patterns used for existence checking in Section IV-B2 and perform logic simulation. We build the truth table of the function h on the divisor set under these PI patterns. Assume that the size of the divisor set is m. Then, the truth table has 2^m input-output pairs. In the truth table, each input is a possible pattern on the divisor set, and the corresponding output denotes the function h's value for that pattern on the divisor set. If a pattern on the divisor set appears in simulation, then the corresponding output in the truth table is set as node n's value under the pattern. Otherwise, if a pattern on the divisor set does not appear in simulation, then this pattern is treated as a *don't care* pattern. Note that since the divisor set is feasible, although multiple PI patterns may produce the same pattern on the divisor set, n's values for all of them are the same.

From the truth table, we can obtain the sum-of-products (SOP) expression of h, which can be done using a two-level logic synthesis tool such as *Espresso* [38]. The resulting SOP expression is then converted to one or more nodes in the circuit, which are used to approximate the original node n, thus simplifying the circuit.

Example 5 *Building on Example 4 and referring to Fig. 3(a),* $\{b, c\}$ *is a feasible divisor set that can approximately resubstitute node* w, *if logic simulation is performed with 5 PI patterns* $abcd = 0100, 0101, 0111, 1011$, *and* 1110. *To derive the new function on* $\{b, c\}$ *for resubstitution, a truth table shown in Table II is built with inputs* b *and* c *and output* $\hat{w}$. *In this simulation, the pattern* $bc = 00$ *does not appear, so it is treated as a don't care pattern. On the other hand, for the patterns* $bc = 01, 10$, *and* 11 *appearing in the simulation, the corresponding output values are set as* $\hat{w} = 1, 1$, *and* 0, *respectively, which can be directly obtained from the shaded entries in Table I. Given this truth table, a possible SOP expression is* $\hat{w} = \overline{bc}$. *This expression is then implemented in the AIG as a node with a complemented output edge, as shown in Fig. 3(c).*

4) Generating Candidate AppResubs: Based on the answers to the three questions in Sections IV-B1–IV-B3, we propose a procedure to generate candidate AppResubs for each node n in the circuit $\mathbb{C}$, as shown in Algorithm 1.

TABLE II: A truth table of an approximate function with inputs b and c and output $\hat{w}$ in Example 5.

bc	00	01	10	11
$\hat{w}$	- (don't care)	1	1	0

Algorithm 1: *GenerateCandidateAppResubs*$(\mathbb{C}, R, L)$

```
Input: Circuit $\mathbb{C}$, user-specified simulation round $R$,
       maximum number of candidate AppResubs $L$
Output: A set of candidate AppResubs $\Psi$
1  Simulate $\mathbb{C}$ for $R$ rounds;
2  The set of candidate AppResubs $\Psi \leftarrow \emptyset$;
   // Add 0-divisor const AppResubs into $\Psi$
3  foreach node $n$ in $\mathbb{C}$ do
4      if $|\Psi| \geq L$ then break;
5      count0 $\leftarrow$ count of 0s in $n$'s simulation values;
6      if count0 $\geq R/2$ then $\Psi \leftarrow \Psi \cup \{Const0Resub(n)\}$;
7      else $\Psi \leftarrow \Psi \cup \{Const1Resub(n)\}$;
   // Add 1-divisor AppResubs into $\Psi$
8  $\Psi \leftarrow AddSomeAppResubs(\Psi, nDiv{=}1, \mathbb{C}, L)$;
   // Add 2-divisor AppResubs into $\Psi$
9  $\Psi \leftarrow AddSomeAppResubs(\Psi, nDiv{=}2, \mathbb{C}, L)$;
10 return $\Psi$;
```

Algorithm 2: *AddSomeAppResubs*$(\Psi, nDiv, \mathbb{C}, L)$

```
Input: The current set of candidate AppResubs $\Psi$,
       number of divisors nDiv, the circuit $\mathbb{C}$ associated
       with simulation results, maximum number of
       candidate AppResubs $L$
Output: The updated set of candidate AppResubs $\Psi$
1  foreach node $n$ in $\mathbb{C}$ do
2      $\mathbb{S} \leftarrow$ divisor sets with nDiv divisors;
3      foreach divisor set $\boldsymbol{g}$ in $\mathbb{S}$ do
4          if $|\Psi| \geq L$ then break;
5          if $\boldsymbol{g}$ is feasible to resubstitute $n$ then
6              New function $\hat{f} \leftarrow BuildFunction(n, \boldsymbol{g})$;
7              if $Resub(n, \boldsymbol{g}, \hat{f})$ can reduce area then
8                  $\Psi \leftarrow \Psi \cup \{Resub(n, \boldsymbol{g}, \hat{f})\}$;
9  return $\Psi$;
```

Algorithm 1 inputs a circuit $\mathbb{C}$, a user-specified simulation round R, and a maximum count of candidate AppResubs L. It outputs a set of candidate AppResubs, denoted as Ψ. The algorithm first performs R rounds of logic simulation on circuit $\mathbb{C}$ (Line 1). Then, it generates 0-divisor constant resubstitutions (Lines 3–7), 1-divisor AppResubs (Line 8), and 2-divisor AppResubs (Line 9) in sequence.

To generate constant resubstitutions, for each node n in the circuit $\mathbb{C}$, we count the number of 0s in n's simulation values across R rounds of simulation (Line 5). If the number is larger than or equal to $R/2$, indicating n aligns more closely with a constant 0, then we add a constant-0 resubstitution into the candidate AppResub set Ψ (Line 6). Otherwise, a constant-1 resubstitution is added (Line 7).

The generation of 1-divisor and 2-divisor AppResubs has a similar process, as shown in Algorithm 2. Specifically, for each node n, we first select some divisor sets with *nDiv* elements, denoted as $\mathbb{S}$, using the method in Section IV-B1 (Line 2). For each divisor set $\boldsymbol{g}$ in $\mathbb{S}$, we check whether $\boldsymbol{g}$ is feasible to approximately resubstitute n using the method in Section IV-B2 (Line 5). If it is feasible, we build an AppResub function $\hat{f}$ for n on $\boldsymbol{g}$ using the method in Section IV-B3 (Line 6). Finally, if this AppResub, denoted as $Resub(n, \boldsymbol{g}, \hat{f})$, can reduce the circuit area (Line 7), then we add it into the candidate AppResub set Ψ (Line 8).

In the worst case, for a circuit $\mathbb{C}$ with N nodes, the potential number of candidate AppResubs is $O(N^3)$. It is because for each node in the circuit, we generate 1 constant resubstitution, no more than N 1-divisor AppResubs, and no more than N^2 2-divisor AppResubs. Thus, the upper bound on the total number of candidate AppResubs for all N nodes in $\mathbb{C}$ is $N \times (1 + N + N^2)$, *i.e.*, $O(N^3)$. For efficiency consideration, we also limit the total number of candidate AppResubs. Line 4 of Algorithm 1 and Line 4 of Algorithm 2 ensure that the number of candidate AppResubs does not exceed L.

V. Resubstitution-Based Efficient ALS Flow

In this section, we propose an efficient ALS flow based on AppResub. We first describe the overall flow in Section V-A. Then, we detail a key step of the flow in Section V-B, *i.e.*, determining a set of *promising* AppResubs.

A. Overall Flow

Our previous work [19] described an ALS flow that iteratively applies the minimum-error AppResub to simplify the circuit, until the error constraint is no longer satisfied. Since the minimum-error AppResub usually increases the error by a small amount, the flow in [19] may require a large number of iterations to reach the error threshold and obtain the final approximate circuit, causing a long runtime.

To accelerate ALS, we propose a new ALS flow in Algorithm 3, featuring two phases:

1) iteratively applying *multiple* promising AppResubs (Algorithm 3 Lines 2–7), and
2) iteratively applying the *single* best AppResub (Algorithm 3 Lines 8–13).

Phase 1) accelerates the convergence of the ALS flow, and Phase 2) aims to further improve the quality of the final approximate circuit.

Algorithm 3: *ResubALS*$(\mathbb{C}_{acc}, E_t, R, L)$.

```
Input: accurate circuit $\mathbb{C}_{acc}$, error threshold $E_t$,
       user-specified simulation round $R$, max number of
       candidate AppResubs $L$
Output: approximate circuit $\mathbb{C}$
1  Approximate circuit $\mathbb{C} \leftarrow \mathbb{C}_{acc}$, error $E \leftarrow 0$;
   // 1)apply MULTIPLE promising AppResubs
2  while true do
3      $\Psi \leftarrow GenerateCandidateAppResubs(\mathbb{C}, R, L)$;
4      Promising resubs. $\Pi^* \leftarrow GetPromAppResubs(\Psi, \mathbb{C})$;
5      $E \leftarrow$ error of an approximate circuit obtained by
         applying to $\mathbb{C}$ all AppResubs in $\Pi^*$;
6      if $E > E_t$ then break;
7      else $\mathbb{C} \leftarrow SimplifyWithMultipleAppResubs(\mathbb{C}, \Pi^*)$;
   // 2)apply SINGLE best AppResub
8  while true do
9      $\Psi \leftarrow GenerateCandidateAppResubs(\mathbb{C}, R, L)$;
10     Best resub. $\pi^* \leftarrow GetSmallestErrorAppResub(\Psi, \mathbb{C})$;
11     $E \leftarrow$ error of an approximate circuit obtained by
         applying $\pi^*$ to $\mathbb{C}$;
12     if $E > E_t$ then break;
13     else $\mathbb{C} \leftarrow SimplifyWithOneAppResub(\mathbb{C}, \pi^*)$;
14 return $\mathbb{C}$;
```

Algorithm 3 inputs an accurate circuit $\mathbb{C}_{acc}$, an error threshold E_t, a user-specified simulation round R for generating candidate AppResubs, and the limit L for the maximum count of candidate AppResubs. It returns an approximate circuit $\mathbb{C}$ with an error no more than E_t. The error metric can be any average error metric, including error rate, mean error distance, mean Hamming distance, and mean square error.

Phase 1) iteratively applies a set of promising AppResubs to simplify the circuit until the error reaches the threshold (Lines 2–7). Each iteration begins with generating a set of candidate AppResubs Ψ (Line 3) using Algorithm 1. From these candidates, a subset of promising AppResubs Π^* is selected (Line 4). The selection method will be detailed in Section V-B. The error E of an approximate circuit obtained by applying to $\mathbb{C}$ all AppResubs in Π^* is computed. If E exceeds E_t (Line 6), the process transitions to Phase 2). If not, all AppResubs in Π^* are applied to the circuit $\mathbb{C}$, followed by traditional logic synthesis to remove redundancy in $\mathbb{C}$ (Line 7).

Phase 2), in contrast to Phase 1), only applies the best AppResub with the smallest error in each iteration (Lines 8–13). Essentially, Phase 2) is similar to the flow in [19], which is not further elaborated here.

B. Determine a Set of Promising AppResubs

In Algorithm 3, Line 4 determines a set of *promising AppResubs*, denoted as Π^*, from the set of candidate AppResubs Ψ. The objective of our flow is to achieve maximum area reduction in the circuit, leading to the formulation of the optimization problem in Eq. (6):

$$\begin{aligned} \max_{\Pi\subseteq\Psi} \quad & \sum_{\pi\in\Pi} \mathit{AreaReduction}(\pi) \\ \text{s.t.} \quad & \textit{1) } \mathit{Error}(\Pi) \le E_t, \\ & \textit{2) } \forall \pi_1,\pi_2 \in \Pi,\ \textit{if } \pi_1 \neq \pi_2, \mathit{Node}(\pi_1) \neq \mathit{Node}(\pi_2), \end{aligned} \tag{6}$$

where Π is a subset of the candidate AppResub set Ψ, with each π being an AppResub in Π. The function $\mathit{AreaReduction}(\pi)$ indicates the area saved when applying an AppResub π to the circuit $\mathbb{C}$. For instance, consider the AIG in Fig. 3(a). After applying an AppResub that resubstitutes node w using $\hat{w} = \overline{b}c$, 4 nodes r, s, u, and v are removed, achieving an area reduction of 4. The function $\mathit{Error}(\Pi)$ computes the error of an approximate circuit obtained by applying all AppResubs in Π to the circuit $\mathbb{C}$. Here, the error can be any average error metric, such as error rate, mean error distance, mean Hamming distance, and mean square error. The function $\mathit{Node}(\pi)$ returns the node resubstituted by AppResub π.

The objective of the problem in Eq. (6) is to maximize the total area reduction by applying all AppResubs in Π to the circuit $\mathbb{C}$. The first constraint guarantees that the error induced by Π is within the threshold E_t. The second constraint ensures that each node in $\mathbb{C}$ is only resubstituted once, thereby preventing any two AppResubs in Π from resubstituting the same node.

In what follows, the method of obtaining $\mathit{Error}(\Pi)$ is introduced in Sections V-B1 and V-B2. Following that, Section V-B3 describes a dynamic-programming-based solution to the problem.

1) Computing the Error for a Set of AppResubs: To determine the error introduced by a set of AppResubs Π, previous studies [32] and [33] used a linear model as follows:

$$\mathit{Error}(\Pi) \approx \mathit{Error}(\mathbb{C}) + \sum_{\pi\in\Pi} \Delta \mathit{Error}(\pi), \tag{7}$$

where $\mathit{Error}(\mathbb{C})$ is the initial error of circuit $\mathbb{C}$ before applying any AppResub in Π, and $\Delta\mathit{Error}(\pi)$ is the *incremental error* caused by individually applying AppResub π to circuit $\mathbb{C}$. Specifically, $\Delta\mathit{Error}(\pi)$ is computed as the difference between the error of an approximate circuit obtained by applying π to $\mathbb{C}$ and $\mathit{Error}(\mathbb{C})$.

However, the linearity assumed in Eq. (7) does not account for the potential interactions among AppResubs, leading to inaccuracies. For instance, in the AIG shown in Fig. 3(a), one AppResub modifying node w impacts the PO y_o, while another AppResub modifying node t could also influence y_o. Such interactions among AppResubs mean that $\mathit{Error}(\Pi)$, could be less than, equal to, or greater than $\mathit{Error}(\mathbb{C}) + \sum_{\pi\in\Pi} \Delta\mathit{Error}(\pi)$, as discussed in [32]. Thus, the error model in Eq. (7) is unsuitable for solving the problem in Eq. (6), since the error constraint $\mathit{Error}(\Pi) \le E_t$ is highly likely to be violated. If the error constraint is violated, then Algorithm 3 will exit Phase 1) and proceed to Phase 2), which only applies the single best AppResub in each iteration, dramatically slowing down the ALS flow.

To avoid violating the error constraint and efficiently estimate $\mathit{Error}(\Pi)$, we establish an *approximate* upper bound of $\mathit{Error}(\Pi)$ by calculating:

$$\mathit{ErrorUpBound}(\Pi) = \mathit{Error}(\mathbb{C}) + \sum_{\pi\in\Pi} \Delta\mathit{ErrorUpBound}(\pi), \tag{8}$$

where $\Delta\mathit{ErrorUpBound}(\pi)$ represents an upper bound of $\Delta\mathit{Error}(\pi)$, *i.e.*, the incremental error caused by an individual AppResub π. Note that we call $\mathit{ErrorUpBound}(\Pi)$ an approximate upper bound, since it is not a strict upper bound but rather an upper bound with a high probability. The details of computing $\mathit{ErrorUpBound}(\pi)$ and the justification for it being an approximate upper bound will be discussed in Section V-B2.

Using Eq. (8), we relax the error constraint of the problem in Eq. (6), leading to the relaxed problem in Eq. (9):

$$\begin{aligned} \max_{\Pi\subseteq\Psi} \quad & \sum_{\pi\in\Pi} \mathit{AreaReduction}(\pi) \\ \text{s.t.} \quad & \textit{1) } \mathit{ErrorUpBound}(\Pi) \le E_t, \\ & \textit{2) } \forall \pi_1,\pi_2 \in \Pi,\ \textit{if } \pi_1 \neq \pi_2, \mathit{Node}(\pi_1) \neq \mathit{Node}(\pi_2). \end{aligned} \tag{9}$$

Clearly, if $\mathit{ErrorUpBound}(\Pi)$ serves as a true upper bound of $\mathit{Error}(\Pi)$, then any solution satisfying the constraints of the relaxed problem must also satisfy those of the problem in Eq. (6). This relaxation not only ensures adherence to the error constraint, but also facilitates the error computation, thereby accelerating the overall ALS flow. However, since $\mathit{ErrorUpBound}(\Pi)$ is an approximate upper bound, in rare cases, $\mathit{Error}(\Pi)$ may exceed $\mathit{ErrorUpBound}(\Pi)$, which may further lead to an invalid solution Π^*_{wrong} to the problem in Eq. (6). Nevertheless, the real error caused by Π^*_{wrong} will be measured and checked by Lines 5–6 of Algorithm 3, deciding whether to apply Π^*_{wrong} or not. This mechanism ensures that Algorithm 3 finally generates an approximate circuit satisfying the error constraint.

2) Computing the Approximate Error Upper Bound: This section shows how to compute an approximate upper bound of $\mathit{Error}(\Pi)$. By Eq. (8), it is based on $\Delta\mathit{ErrorUpBound}(\pi)$. Thus, we first describe how to compute $\Delta\mathit{ErrorUpBound}(\pi)$ for an AppResub π in the circuit $\mathbb{C}$.

We first build an approximation miter with the accurate circuit $\mathbb{C}_{acc}$ and the approximate circuit $\mathbb{C}$, as shown in Fig. 2. As mentioned in Section II-C, the miter encodes the deviation function D between $\mathbb{C}_{acc}$ and $\mathbb{C}$ using K bits $d_1, d_2, \ldots, d_K$. Notably, the approximation miter can compute the error of $\mathbb{C}$ for any average error metric, including error rate, mean error distance, mean Hamming distance, and mean square error. Recall that Eq. (5) utilizes the approximation miter to compute the error of $\mathbb{C}$ as $\mathit{Error}(\mathbb{C}) = \frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{K} 2^{k-1} d_k(\boldsymbol{x}^i)$.

Assume that after applying AppResub π to circuit $\mathbb{C}$, the resulting approximate circuit is $\mathbb{C}'$. We also assume that the miter including $\mathbb{C}_{acc}$ and $\mathbb{C}'$ has K outputs $d'_1, d'_2, \ldots, d'_K$. By Eq. (5), $\mathit{Error}(\mathbb{C}')$ can be computed similarly. The *incremental error* caused by applying AppResub π to circuit $\mathbb{C}$, denoted as $\Delta\mathit{Error}(\pi)$, is then expressed as:

$$\begin{aligned}\Delta\mathit{Error}(\pi) &= \mathit{Error}(\mathbb{C}') - \mathit{Error}(\mathbb{C}) \\ &= \frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{K} 2^{k-1}\left(d'_k(\boldsymbol{x}^i) - d_k(\boldsymbol{x}^i)\right) \\ &= \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i=1}^{M}\left(d'_k(\boldsymbol{x}^i) - d_k(\boldsymbol{x}^i)\right)\right).\end{aligned} \tag{10}$$

Considering $d_k(\boldsymbol{x}^i)$ and $d'_k(\boldsymbol{x}^i)$ can only be 0 or 1, we further have:

$$\begin{aligned}&\Delta\mathit{Error}(\pi) = \\ &\frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i:d_k(\boldsymbol{x}^i)=0} d'_k(\boldsymbol{x}^i) + \sum_{i:d_k(\boldsymbol{x}^i)=1}\left(d'_k(\boldsymbol{x}^i) - 1\right)\right) \\ &\le \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i:d_k(\boldsymbol{x}^i)=0} d'_k(\boldsymbol{x}^i)\right).\end{aligned} \tag{11}$$

Thus, we let

$$\Delta\mathit{ErrorUpBound}(\pi) = \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i:d_k(\boldsymbol{x}^i)=0} d'_k(\boldsymbol{x}^i)\right), \tag{12}$$

which serves as an upper bound of $\Delta\mathit{Error}(\pi)$. To efficiently compute $d'_k(\boldsymbol{x}^i)$ for each k and i, we utilize the method based on the *change propagation matrix* in [29].

By substituting $\Delta\mathit{ErrorUpBound}(\pi)$ from Eq. (12) into Eq. (8), we can compute $\mathit{ErrorUpBound}(\Pi)$. Now, we explain why $\mathit{ErrorUpBound}(\Pi)$ is an *approximate upper bound* of $\mathit{Error}(\Pi)$ (the real error caused by Π).

Assume that $\Pi = \{\pi_1, \ldots, \pi_S\}$, where π_j represents a specific AppResub in the set of AppResubs Π. Based on Constraint 2 in the problems in Eqs. (6) and (9), we can assume that each π_j targets a unique node for resubstitution. After applying all π_j's in Π to the circuit $\mathbb{C}$, the resulting approximate circuit is denoted as $\mathbb{C}^{\Pi}$, with the miter outputs being $d_1^{\Pi}, d_2^{\Pi}, \ldots, d_K^{\Pi}$. Similar to Eqs. (10) and (11), the error after applying all AppResubs in Π is computed as:

$$\begin{aligned}&\mathit{Error}(\Pi) = \mathit{Error}(\mathbb{C}) + \Delta\mathit{Error}(\Pi) \\ =&\mathit{Error}(\mathbb{C}) + \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i=1}^{M}\left(d_k^{\Pi}(\boldsymbol{x}^i) - d_k(\boldsymbol{x}^i)\right)\right) \\ \le&\mathit{Error}(\mathbb{C}) + \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i:d_k(\boldsymbol{x}^i)=0} d_k^{\Pi}(\boldsymbol{x}^i)\right).\end{aligned} \tag{13}$$

Assume that after applying AppResub π_j to circuit $\mathbb{C}$, the resulting approximate circuit is $\mathbb{C}^{\pi_j}$, and the corresponding miter outputs are $d_1^{\pi_j}, d_2^{\pi_j}, \ldots, d_K^{\pi_j}$. By Eqs. (8) and (12), we have

$$\begin{aligned}&\mathit{ErrorUpBound}(\Pi) \\ =&\mathit{Error}(\mathbb{C}) + \frac{1}{M}\sum_{k=1}^{K} 2^{k-1}\left(\sum_{i:d_k(\boldsymbol{x}^i)=0}\left(\sum_{\pi_j\in\Pi} d_k^{\pi_j}(\boldsymbol{x}^i)\right)\right).\end{aligned} \tag{14}$$

Comparing Eqs. (13) and (14), to prove that $\mathit{ErrorUpBound}(\Pi)$ is an approximate upper bound of $\mathit{Error}(\Pi)$, we only need to show that for all k's and i's satisfying $d_k(\boldsymbol{x}^i) = 0$, it is very likely that $d_k^{\Pi}(\boldsymbol{x}^i) \le \sum_{\pi_j\in\Pi} d_k^{\pi_j}(\boldsymbol{x}^i)$.

Since $d_k^{\Pi}(\boldsymbol{x}^i)$ and $d_k^{\pi_j}(\boldsymbol{x}^i)$ can only be 0 or 1, we only need to consider the case where $d_k^{\Pi}(\boldsymbol{x}^i) = 1$. When $d_k(\boldsymbol{x}^i) = 0$, the assumption that $d_k^{\Pi}(\boldsymbol{x}^i) = 1$ means that under the PI pattern $\boldsymbol{x}^i$, the application of all AppResubs in Π influences the miter output d_k, changing d_k's value from 0 to 1. Similarly, if $d_k(\boldsymbol{x}^i) = 0$ and $d_k^{\pi_j}(\boldsymbol{x}^i) = 1$, this indicates that the application of π_j influences the miter output d_k, changing d_k's value from 0 to 1. In essence, if any single AppResub π_j in Π influences d_k, then the simultaneous application of all AppResubs in Π will also influence d_k with a high probability.

Formally, this relationship can be expressed as

$$d_k^{\Pi}(\boldsymbol{x}^i) \approx d_k^{\pi_1}(\boldsymbol{x}^i) \vee d_k^{\pi_2}(\boldsymbol{x}^i) \vee \ldots \vee d_k^{\pi_S}(\boldsymbol{x}^i) \le \sum_{\pi_j\in\Pi} d_k^{\pi_j}(\boldsymbol{x}^i),$$

where $\vee$ denotes the OR operation, and the inequality is because the OR of a set of binary variables is no more than their sum. The inequality means that for all k's and i's satisfying $d_k(\boldsymbol{x}^i) = 0$, $d_k^{\Pi}(\boldsymbol{x}^i) \le \sum_{\pi_j\in\Pi} d_k^{\pi_j}(\boldsymbol{x}^i)$ approximately holds. Therefore, $\mathit{Error}(\Pi) \le \mathit{ErrorUpBound}(\Pi)$ also approximately holds.

3) Knapsack-Based Solution: The problem in Eq. (9) can be formulated as a specific knapsack problem as follows.

- Elements:
  - *Item*: each AppResub π in the candidate AppResub set Ψ corresponds to an item.
  - *Value*: the area reduction caused by an AppResub π, denoted as $\mathit{AreaReduction}(\pi)$, corresponds to the value of the item.
  - *Weight*: the approximate upper bound of the incremental error caused by an AppResub π, denoted as $\Delta\mathit{ErrorUpBound}(\pi)$, corresponds to the *weight* of the item.
- Objective: maximizing the total value (area reduction) of the items selected for the knapsack.
- Constraints:
  - *Capacity constraint* (corresponding to Constraint 1 of the problem in Eq. (9)): the error margin $E_{margin} = E_t - \mathit{Error}(\mathbb{C})$ is set as the knapsack capacity.
  - *Selection constraint* (corresponding to Constraint 2 of the problem in Eq. (9)): Assume that circuit $\mathbb{C}$ consists of N nodes $n_1, n_2, \ldots, n_N$. The candidate AppResubs in Ψ are categorized into N groups, $\Psi_1, \Psi_2, \ldots, \Psi_N$, where each Ψ_i contains all candidate AppResubs for node n_i. From each

group of items (*i.e.*, Ψ_i), at most one item (*i.e.*, one AppResub) can be selected and put into the knapsack.

Although this knapsack model is similar to the one in [20], our method significantly diverges from [20] in several ways: First, unlike the method in [20], which is limited to the error rate metric, our approach supports any average error metric, such as error rate, mean error distance, mean Hamming distance, and mean square error. Second, we introduce a more precise model (*i.e.*, Eqs. (8) and (12)) to estimate an error upper bound for a set of AppResubs. Lastly, rather than solving the knapsack problem as in [20], we propose a more efficient solution by addressing its dual problem, which is described next.

A classical method to solve the knapsack problem is based on dynamic programming with the state-transition equation shown in Eq. (15):

$$\begin{aligned}&\mathit{DP}(i,w)=\max\{\mathit{DP}(i-1,w),\\&\mathit{DP}(i-1,w-\Delta \mathit{ErrorUpBound}(\psi_{i,1}))+\mathit{AreaReduction}(\psi_{i,1}),\\&\mathit{DP}(i-1,w-\Delta \mathit{ErrorUpBound}(\psi_{i,2}))+\mathit{AreaReduction}(\psi_{i,2}),\\&\ldots,\\&\mathit{DP}(i-1,w-\Delta \mathit{ErrorUpBound}(\psi_{i,|\Psi_i|}))+\mathit{AreaReduction}(\psi_{i,|\Psi_i|})\},\end{aligned} \tag{15}$$

where $\mathit{DP}(i,w)$ denotes the maximum total value achievable with the first i groups under the capacity w, and $\psi_{i,j}$ is the j-th item in the i-th group Ψ_i. The term $\mathit{DP}(i-1,w)$ accounts for not selecting any item from group i, while $\mathit{DP}(i-1,w-\Delta \mathit{ErrorUpBound}(\psi_{i,j}))+\mathit{AreaReduction}(\psi_{i,j})$ represents selecting the j-th item from group i. This equation aligns with the selection constraint, ensuring at most one item from each group is chosen for the knapsack. The final solution to the knapsack problem is $\mathit{DP}(N,E_{\mathit{margin}})$, where N is the number of groups and E_{margin} is the knapsack's capacity.

A critical issue in solving the knapsack problem using Eq. (15) is the non-integer nature of item weights, which represents the error upper bounds of the AppResubs. Even if we multiply all the weights by a large constant α to make them integers, as proposed in [20], both the time and space complexity of the dynamic-programming-based method will also increase significantly, where the time complexity is $O(\alpha N E_{\mathit{margin}})$. To address this issue, we propose to solve the dual problem of the knapsack problem using the state-transition equation shown in Eq. (16):

$$\begin{aligned}&\mathit{DP'}(i,v)=\min\{\mathit{DP'}(i-1,v),\\&\mathit{DP'}(i-1,v-\mathit{AreaReduction}(\psi_{i,1}))+\Delta \mathit{ErrorUpBound}(\psi_{i,1}),\\&\mathit{DP'}(i-1,v-\mathit{AreaReduction}(\psi_{i,2}))+\Delta \mathit{ErrorUpBound}(\psi_{i,2}),\\&\ldots,\\&\mathit{DP'}(i-1,v-\mathit{AreaReduction}(\psi_{i,|\Psi_i|}))+\Delta \mathit{ErrorUpBound}(\psi_{i,|\Psi_i|})\},\end{aligned} \tag{16}$$

where $\mathit{DP'}(i,v)$ denotes the minimum total weight (error upper bound) achievable with the first i groups to obtain a value of at least v (total area reduction). The dual problem's advantage lies in the typical integer nature of item values (area reduction). For example, if our flow works on AIGs, the area reduction (value) caused by an AppResub (item) is the number of nodes removed, inherently an integer. By avoiding the non-integer issue, the shift to the dual problem significantly reduces the time and space complexity. The solution to the dual problem is denoted as V^*, which is the largest integer satisfying $\mathit{DP'}(N,V^*)\le E_{\mathit{margin}}$. It can be proven that the original knapsack problem has the same solution as its dual problem, *i.e.*, $\mathit{DP}(N,E_{\mathit{margin}})=V^*$. From the solution to the dual problem, we can easily recover the selected items in the knapsack. These selected items (*i.e.*, AppResubs) can further construct the solution to the problem in Eq. (9), *i.e.*, the set of promising AppResubs Π^*. Note that sometimes there does not exist any Π satisfying the error constraint of the problem in Eq. (9). In this case, Algorithm 3 terminates Phase 1), *i.e.*, iteratively applying multiple promising AppResubs, and moves to Phase 2), *i.e.*, iteratively applying the single best AppResub.

VI. Experimental Results

A. Experimental Setup

Our resubstitution-based ALS flow is implemented in C++ and evaluated on a laptop with AMD Ryzen9 7945HX processors and 64GB RAM. To boost our method's efficiency, our flow is parallelized using 32 threads, particularly focusing on the most time-consuming steps in Algorithm 3: *GetPromAppResubs* and *GetSmallestErrorAppResub*. These steps involve estimating the errors for all candidate AppResubs, with each thread tasked with computing the error estimation for a specific candidate AppResub.

In all experiments, our ALS flow begins by converting the original circuit into an AIG and applies AppResubs to the AIG. The reason for using AIGs is that many AIG-based logic synthesis works [35], [39] and ALS works [16], [33] have shown significant advantages in reducing hardware cost, particularly for CMOS technologies. After simplifying the original AIG with our ALS flow, the resulting approximate AIG is mapped into a gate netlist using ABC [40]. Unless otherwise specified, the standard cell library employed is the Nangate 45nm library [41]. Furthermore, as mentioned in Section V-A, the *SimplifyWithMultipleAppResubs* function in Algorithm 3 not only simplifies the circuit by applying AppResubs, but also performs traditional logic synthesis to further remove circuit redundancy. Since our goal is reducing area, the area-oriented optimization script *"compress2rs"* in ABC is applied. Given the randomness in logic simulation, our ALS flow may produce different approximate circuits in different runs. Thus, all experiments on our method are performed three times, and the circuit with the smallest area is reported. Meanwhile, the runtime of our method is reported as the total runtime of the three runs.

To evaluate the hardware cost of a circuit, we utilize *area ratio* (the area of the approximate circuit over that of the accurate one) and *delay ratio* (the delay of the approximate circuit over that of the accurate one). Smaller ratios are preferred due to more reduction in area and delay. Unless otherwise specified, in all experiments with our ALS flow and other ALS flows, the area and delay of circuits are measured after technology mapping using ABC. To evaluate the accuracy of circuits, four different average error metrics, error rate, normalized mean error distance, normalized mean Hamming distance, and mean square error are considered in our experiments. They are measured by performing 102,400 rounds of logic simulation to ensure accuracy. Note that our method only supports average errors, so we do not compare it with other ALS methods under the maximum error constraints, such as [26] and [27].

The benchmarks used in our experiments are listed in Table III. They are accurate circuits selected from ISCAS85 [42], BACS [43], and EPFL [44] benchmarks. These circuits are used in the related works that we compare against [18], [25], [45]. Table III reports the sizes and depths of the benchmarks in AIGs. As in [45], the AIGs have been well optimized to ensure as little redundancy as possible. These AIGs are then used as inputs to our ALS flow and those ALS flows for comparison. Table III also includes the areas and delays of the benchmarks after mapping the optimized AIGs using the Nangate 45nm library.

In the following, we first describe an experiment used to choose parameters of our ALS flow. Then, we will compare our ALS flow with state-of-the-art methods under different error metrics, *i.e.*, error rate, normalized mean error distance, normalized mean Hamming distance, and mean square error.

TABLE III: Experimental benchmarks. Area and delay are measured by mapping the AIGs into the Nangate 45nm library.

Benchmark suite	Circuit	#PIs/#POs	AIG		Gate netlist	
			Size	Depth	Area/μm^2	Delay/ns
ISCAS85	c880	60/26	313	22	198.17	0.59
	c1355	41/32	390	16	235.94	0.56
	c1908	33/25	367	25	229.56	0.86
	c2670	233/140	579	17	385.17	0.68
	c3540	50/22	937	32	521.09	1.02
	c5315	178/123	1306	28	720.33	0.72
	c7552	207/108	1469	26	903.60	1.43
BACS	absdiff	16/9	104	14	70.22	0.41
	add32	64/33	302	20	211.20	0.55
	buttfly	32/34	226	31	153.75	1.05
	mac	12/8	124	20	90.97	0.59
	mult8	16/16	470	44	336.76	1.30
	mult16	32/32	2033	41	1445.98	1.29
EPFL	add128	256/129	1019	314	982.9	4.83
	barshift	135/128	2688	14	1945.0	0.85
	div	128/128	23667	4473	19949.5	89.78
	log2	32/32	38540	419	26422.8	11.56
	max	512/130	2686	549	2456.2	10.98
	mult64	128/128	33242	326	22401.5	6.87
	sine	24/25	7044	180	5334.4	4.50
	sqrt	128/64	21951	4591	19035.5	128.16
	square	64/128	20030	296	14394.6	5.88

B. Parameter Choices

Our ALS flow, as detailed in Algorithm 3, has two important parameters: the number of simulation rounds R for generating the candidate AppResubs and the maximum number of candidate AppResubs L. We design the following experiment on the ISCAS85 benchmarks for choosing these parameters. The applied error metric is error rate, set with a threshold of 5%.

First, we do not limit the total number of candidate AppResubs ($L = +\infty$) and observe the impact of varying R on the number of AppResubs, the area ratio of final approximate circuits, and the runtime. The upper part of Fig. 4 plots the number of candidate AppResubs in the first iteration versus R. It is observed that increasing R generally results in a decrease in the number of candidate AppResubs. This is reasonable because with a larger R, the truth tables of the resubstitution functions are built

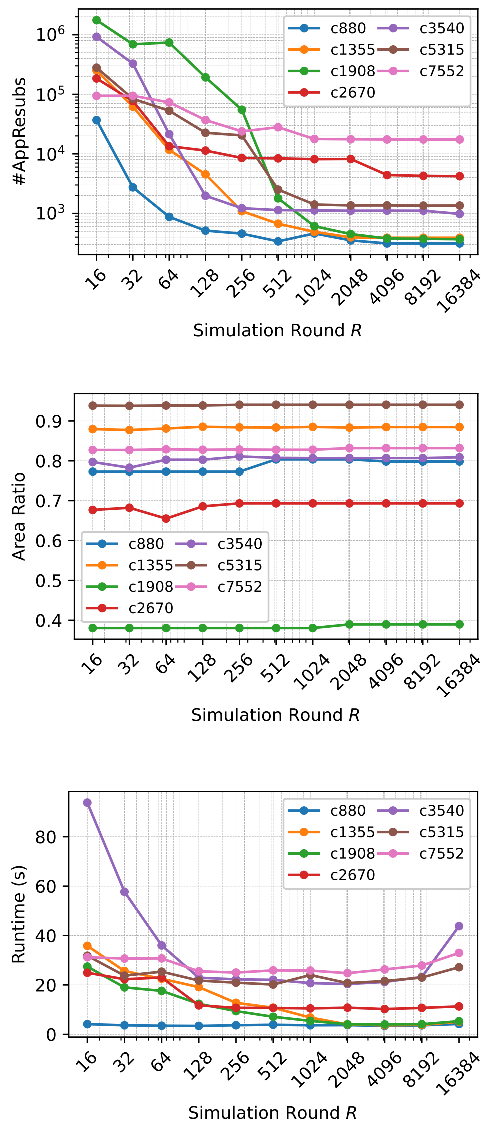


Fig. 4: Impact of the number of simulation rounds R on the number of candidate AppResubs in the first iteration, the area ratio of the final approximate circuits, and the total runtime of generating the final approximate circuits.

with more PI patterns, and the resubstitution functions are closer to the accurate ones. This narrows the space for approximation and thus reduces the number of candidate AppResubs. However, exceptions do exist, such as the case of *c1908* when R increases from 32 to 64. This is caused by the randomness of logic simulation.

The middle part of Fig. 4 plots the area ratios of the final approximate circuits versus R. The area ratios remain relatively stable for different R's. Notably, for all benchmarks, the area ratios achieved by a smaller R are slightly smaller than those with a larger R. For example, for benchmarks *c880, c2670*, and *c3540*, the area ratios achieved by $R = 16$, 32, and 64 are obviously smaller than those achieved by $R = 8192$ and 16384. As analyzed above, a smaller R means more candidate AppResubs, and thus our ALS flow explores a larger solution space and has more opportunities to find better approximate circuits with smaller area ratios.

In the bottom part of Fig. 4, the total runtime of generating the final approximate circuits is plotted. As R increases, the runtime for the smallest benchmarks *c880* first slightly decreases and then remains stable, and the runtime for the small benchmarks *c1355*, *c1908*, *c2670* shows an overall decreasing trend. This is because a larger R leads to fewer candidate AppResubs, which requires less time to estimate the errors caused by the AppResubs, hence reducing the runtime of the whole ALS flow. Conversely, for larger benchmarks *c3540*, *c5315*, and *c7552*, the runtime initially decreases but then increases with R. The reason that R decreases initially is similar to that for the small benchmarks. However, as R further increases, additional rounds of simulation are required to generate the candidate AppResubs. The simulation time for generating the candidate AppResubs becomes significant and impacts overall runtime, and hence, the runtime of the whole ALS flow increases when R is large.

Given the above observations, to make the ALS flow efficient and effective, R should neither be too small (to avoid long runtime) nor too large (to avoid long runtime and bad quality). In practice, we choose $R = 64$ to balance the final circuit quality and runtime. Meanwhile, we set $L = 10^5$ to limit the total number of candidate AppResubs. This is guided by the observation that the number of candidate AppResubs in the first iteration is less than 10^5 for most benchmarks when $R = 64$. In the following experiments, we choose $R = 64$ and $L = 10^5$ as the default parameters for our ALS flow.

C. Experiments Under the Error Rate Constraint

This set of experiments approximates the ISCAS85 benchmarks under the error rate constraint. Note that these benchmarks are random or control circuits, and error rate is a suitable error metric for them.

1) Comparison with the ALS Flow in [25]: Tam *et al.* [25] proposed an area-oriented ALS flow working on AIGs. We compare our flow to it using a 5% error rate threshold, which is the same threshold used in [25]. The comparison focuses on the sizes of the resulting approximate AIGs. The data from [25] is used for a direct comparison. The benchmark *c1355* is excluded, since it is not used in [25]. As illustrated in Table IV, our flow consistently achieves smaller approximate AIGs than Tam *et al.*'s flow for all benchmarks. On average, our flow reduces the AIG size ratio by 8.5%, compared to Tam *et al.*'s. Moreover, our flow can generate approximate AIGs within 41 seconds for all benchmarks. Table IV also shows the runtime of Tam *et al.*'s flow for reference, which is directly obtained from [25]. We do not compare the depths of the final AIGs generated by the two flows, since the depths of the final AIGs are not reported in [25]. However, our flow guarantees that the depths of the final AIGs do not exceed those of the original AIGs. Technology mapping results are not included here due to the lack of relevant data in [25], but such results will be presented in subsequent experiments.

2) Comparison with the BLASYS Flow in [18]: Ma *et al.* [18] developed an ALS flow known as *BLASYS* [18], which uses Boolean matrix factorization to reduce circuit area. We run BLASYS's source code and compare it with our flow under error rate thresholds of 0.5% and 5%. For fairness, both the BLASYS flow and our flow utilize 32 CPU threads and apply the same ABC script for synthesis (*"compress2rs"*) and mapping (*"dch;amap"*). Table V compares the results with **bold** entries indicating when our flow outperforms BLASYS.

TABLE IV: Comparison of our flow with that in [25] under the error rate threshold of 5%. The **bold** entries indicate that our flow outperforms that in [25].

Circuit	Original AIG size	AIG size		AIG size ratio[*]		Runtime/s	
		Ours	Tam *et al.*	Ours	Tam *et al.*	Ours	Tam *et al.*
c880	313	**246**	267	**78.6%**	85.3%	4.70	3.71
c1908	367	**152**	161	**41.4%**	43.9%	9.72	3.41
c2670	579	**507**	540	**87.6%**	93.3%	12.38	6.35
c3540	937	**706**	832	**75.3%**	88.8%	35.49	30.79
c5315	1306	**1235**	1328	**94.6%**	101.7%	27.38	9.41
c7552	1469	**1196**	1425	**81.4%**	97.0%	40.11	29.52
Average				**76.5%**	85.0%	21.63	13.87

[*] AIG size ratio = AIG size / Original AIG size.

TABLE V: Comparison of our flow with the BLASYS flow under error rate thresholds of 0.5% and 5%. The **bold** entries indicate that our flow outperforms BLASYS.

Circuit	Error rate bound	Area ratio		Delay ratio[*]		Runtime/s	
		Ours	BLASYS	Ours	BLASYS	Ours	BLASYS
c880	0.5%	**82.3%**	84.3%	**112.2%**	112.6%	**3.9**	234.5
	5%	**76.4%**	78.5%	**101.5%**	108.5%	**3.3**	255.9
c1355	0.5%	**96.3%**	100.1%	**92.5%**	102.6%	**18.3**	153.9
	5%	87.0%	86.2%	**105.4%**	105.8%	**21.3**	262.7
c1908	0.5%	**93.3%**	94.6%	**78.7%**	93.8%	**22.8**	110.3
	5%	**38.0%**	39.2%	42.1%	39.5%	**9.3**	394.4
c2670	0.5%	**67.3%**	71.3%	**89.7%**	98.4%	**10.5**	800.9
	5%	**63.2%**	68.1%	**83.8%**	86.3%	**12.3**	1010.6
c3540	0.5%	**97.1%**	97.5%	99.3%	97.7%	**21.3**	1256.4
	5%	**78.0%**	82.3%	**99.7%**	101.7%	**35.4**	2577.3
c5315	0.5%	**97.9%**	99.2%	**110.3%**	114.7%	**19.8**	3141.4
	5%	**92.5%**	98.0%	**100.6%**	116.1%	**27.3**	4116.2
c7552	0.5%	**81.7%**	84.5%	139.5%	135.9%	**28.5**	4543.9
	5%	**80.8%**	83.1%	**112.9%**	134.0%	**40.2**	5247.3
Average		**80.8%**	83.4%	**97.7%**	103.4%	**19.5**	1721.8

[*] Delay ratios larger than 100% mean delay increase. This is also applicable to the following tables.

For all cases except *c1355* at a 5% error rate threshold, our flow reduces more area than BLASYS. On average, our flow generates approximate circuits with an area ratio of 80.8%, improving over BLASYS by 2.6%. In addition, our flow typically generates approximate circuits with shorter delays than BLASYS. On average, our flow achieves a 5.7% delay reduction over BLASYS. However, our flow may produce approximate circuits with larger delays than the original circuits, as in the case of benchmarks *c880, c1355, c5315,* and *c7552*. This happens because our approach prioritizes area reduction, potentially at the expense of increased delay. In terms of runtime, our flow demonstrates significant efficiency, consistently outperforming BLASYS. On average, it can approximate these benchmarks with only 19.5 seconds, which is $88\times$ faster than BLASYS.

D. Experiments Under the Normalized Mean Error Distance Constraint

This experiment approximates the BACS benchmarks under the normalized mean error distance constraint. Given that these benchmarks are arithmetic circuits, normalized mean error distance, which considers the significance of the circuit outputs, is a suitable error metric for them. Our flow is compared against BLASYS under the normalized mean error distance thresholds of 0.59% and 2.94%. The setup of BLASYS remains the same as that outlined in Section VI-C2.

The comparison result is shown in Table VI. We can see that our flow always achieves larger area reduction than BLASYS. On average, our method produces approximate circuits with an area ratio of 21.9%, showing a 20.9% improvement over BLASYS. Moreover, except for the benchmark *mac* at the 0.59% normalized mean error distance threshold, our flow can generate approximate circuits with smaller delay ratios than BLASYS. On average, our flow reduces 21.7% delay over BLASYS. In terms of runtime, our method only needs 16.2 seconds on average. It significantly outperforms BLASYS and is $400\times$ faster.

TABLE VI: Comparison of our flow with the BLASYS flow under normalized mean error distance (NMED) thresholds of 0.59% and 2.94%. The **bold** entries indicate that our flow outperforms BLASYS.

Circuit	NMED bound	Area ratio		Delay ratio		Runtime/s	
		Ours	BLASYS	Ours	BLASYS	Ours	BLASYS
absdiff	0.59%	**48.5%**	73.9%	**92.9%**	144.9%	**2.7**	29.3
	2.94%	**22.7%**	47.0%	**52.2%**	76.2%	**2.4**	29.0
adder32	0.59%	**10.1%**	32.2%	**45.5%**	68.3%	**8.7**	223.8
	2.94%	**5.8%**	25.7%	**27.1%**	37.5%	**7.2**	261.4
buttfly	0.59%	**20.6%**	50.0%	**37.9%**	53.5%	**6.3**	87.8
	2.94%	**13.3%**	40.5%	**26.7%**	55.1%	**6.0**	95.3
mac	0.59%	**76.6%**	95.6%	106.5%	91.3%	**2.1**	10.6
	2.94%	**23.7%**	46.5%	**37.9%**	63.3%	**2.4**	33.2
mult8	0.59%	**24.9%**	51.6%	**65.2%**	75.4%	**17.1**	845.2
	2.94%	**8.4%**	23.2%	**27.9%**	58.5%	**9.9**	954.6
mult16	0.59%	**7.4%**	16.4%	**49.9%**	67.2%	**69.3**	37383.1
	2.94%	**1.5%**	11.6%	**18.6%**	56.9%	**60.3**	37860.0
Average		**21.9%**	42.8%	**49.0%**	70.7%	**16.2**	6484.5

E. Experiments Under the Normalized Mean Hamming Distance Constraint

This experiment approximates the EPFL benchmarks under the normalized mean Hamming distance constraint. We compare our flow with BLASYS under the normalized mean Hamming distance thresholds of 5% and 10%, which are the same thresholds used in [18]. We do not run the source code of BLASYS in this experiment, since the benchmarks are too large and the runtime of BLASYS is extremely long. Instead, we directly use the data reported in [18] for comparison. Although they are not obtained using the same Nangate 45nm library as our flow, the relative area and delay ratios can still provide a reference on the performance of our flow. Given that the EPFL benchmarks are large, we limit the total number of candidate AppResubs as $L = 20000$ in this experiment, while R is still set as 64.

The comparison result is shown in Table VII. We can see that our flow consistently outperforms BLASYS in area reduction. On average, our flow achieves an area ratio of 59.1%, improving over BLASYS by 26.5%. Particularly, for the benchmark *div* under the 5% normalized mean Hamming distance threshold, our flow reduces the area by 80.1% over BLASYS. Meanwhile, our flow can generate approximate circuits with smaller delay ratios than BLASYS for the benchmarks *div* and *max*. However, we also notice that our flow may produce approximate circuits with larger delays than the original circuits. Moreover, our flow is significantly faster than BLASYS. We can approximate these benchmarks under the normalized mean Hamming distance constraint in 16275 seconds on average.

F. Experiments Under the Mean Square Error Constraint

This experiment approximates some benchmarks from the BACS and EPFL suites under the mean square error

TABLE VII: Comparison of our flow with the BLASYS flow under normalized mean Hamming distance (NMHD) thresholds of 5% and 10%. The **bold** entries indicate that our flow outperforms BLASYS. N/A indicates that the data is not reported in [18].

Circuit	NMHD bound	Area ratio		Delay ratio		Runtime/s	
		Ours	BLASYS	Ours	BLASYS	Ours	BLASYS
adder	5%	**77.0%**	89.4%	106.1%	90.8%	66	20418
	10%	**70.0%**	79.4%	96.7%	80.9%	147	N/A
bar	5%	**87.1%**	95.8%	122.6%	105.6%	84	210600
	10%	**82.5%**	90.0%	93.4%	88.5%	186	N/A
div	5%	**5.8%**	85.9%	**9.8%**	91.6%	7332	N/A
	10%	**1.4%**	76.2%	**1.2%**	73.0%	10479	N/A
log2	5%	**71.3%**	92.9%	109.9%	100.5%	70671	N/A
	10%	**68.9%**	82.1%	105.4%	78.2%	43317	N/A
max	5%	**21.0%**	91.0%	**40.3%**	114.3%	414	N/A
	10%	**18.5%**	77.6%	**12.3%**	94.3%	567	N/A
mult64	5%	**75.2%**	87.7%	117.0%	99.4%	1164	N/A
	10%	**68.8%**	80.5%	113.1%	93.8%	113984	N/A
sin	5%	**63.1%**	84.3%	102.8%	93.1%	1242	1670958
	10%	**62.1%**	71.7%	99.8%	79.9%	2310	N/A
sqrt	5%	46.1%	N/A	79.3%	N/A	2673	N/A
	10%	44.1%	N/A	75.6%	N/A	5085	N/A
square	5%	**88.3%**	95.8%	106.6%	85.8%	3606	N/A
	10%	**85.3%**	88.5%	91.8%	75.5%	5046	N/A
Average w/o sqrt		**59.1%**	85.6%	**83.1%**	90.3%	16275	N/A

TABLE VIII: Comparison of our flow with the DASALS flow under mean square error (MSE) constraint. The MSE bounds are exactly the same as the ones used in [45]. The **bold** entries indicate that our flow outperforms DASALS.

Circuit	MSE bound	Area ratio		Delay ratio		Runtime/s
		Ours	DASALS	Ours	DASALS	Ours
absdiff	33.5	**34.9%**	54.7%	**54.7%**	61.2%	5.4
mac	11.0	**62.6%**	64.3%	**64.3%**	68.2%	3.9
mult8	40.0	**82.8%**	87.5%	87.5%	68.4%	49.2
adder32	48.2	**79.9%**	93.4%	93.4%	51.4%	1047.3
sin	25.7	**95.7%**	98.4%	**98.4%**	100.4%	4758.9
Average		**71.2%**	79.6%	79.6%	69.9%	1173.0

constraint. The compared ALS flow is DASALS [45]. DASALS formulates the ALS problem as a differentiable architecture search problem. It tries to directly search the whole circuit structure to generate better approximate circuits. Only part of the benchmarks in the BACS and EPFL suites are tested in [45], and we select them for comparison. Note that the work [45] only tests the mean square error constraint and does not report the performance under other error metrics, so we only compare our flow with DASALS under the mean square error constraint. We directly use the data reported in [45] and maintain the same mean square error bounds as those used in [45] for a fair comparison. Moreover, both our flow and DASALS utilize the MCNC standard cell library [46] for technology mapping.

As shown in Table VIII, our flow outperforms DASALS in terms of area ratios for all benchmarks, and reduces more delays than DASALS for all but two benchmarks, *mult8* and *adder32*. On average, our flow achieves an 8.4% area reduction, compared to DASALS. Moreover, we can approximate these benchmarks under the mean square error constraint in 1173 seconds on average. Table VIII omits the runtime of DASALS since [45] does not report it.

VII. Conclusion

This work proposes an efficient resubstitution-based ALS flow. The proposed flow is based on an effective LAC called AppResub, which approximately simplifies a circuit by re-expressing a node's function using a set of other nodes in the circuit. We design a simulation-based method to efficiently generate candidate AppResubs in a circuit. Furthermore, we design a two-phase ALS flow. The first phase iteratively applies multiple promising AppResubs to accelerate the ALS flow, while the second further improves the circuit quality by iteratively applying the AppResub with the smallest error. To determine a set of promising AppResubs in the first phase, we formulate a problem to maximize the area reduction while satisfying the error constraint. It is solved by dynamic programming on a relaxed problem using a proposed error upper bound model. Experiments show that our flow is efficient and significantly reduces the hardware cost of resulting approximate circuits.

Acknowledgement

This work is supported by the Swiss National Science Foundation Grant "Supercool: Design methods and tools for superconducting electronics" with funding number 200021_1920981, by the Synopsys Inc., and by the National Key R&D Program of China under Grant 2021ZD0114701. We thank the anonymous reviewers for their constructive feedback.

References

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

[2] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[3] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[4] R. Ye, T. Wang, *et al.*, "On reconfiguration-oriented approximate adder design and its application," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 48–54.

[5] Y. Kim *et al.*, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 130–137.

[6] J. Hu and W. Qian, "A new approximate adder with low relative error and correct sign calculation," in *Design, Automation and Test in Europe (DATE)*, 2015, pp. 1449–1454.

[7] V. Camus, M. Cacciotti, *et al.*, "Design of approximate circuits by fabrication of false timing paths: The carry cut-back adder," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 746–757, 2018.

[8] C. Liu *et al.*, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Design, Automation & Test in Europe (DATE)*, 2014, 95:1–95:4.

[9] S. Rehman, W. EI-Harouni, *et al.*, "Architectural-space exploration of approximate multipliers," in *International Conference on Computer-Aided Design (ICCAD)*, 2016, 80:1–80:8.

[10] M. S. Ansari, B. F. Cockburn, and J. Han, "A hardware-efficient logarithmic multiplier with improved accuracy," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 928–931.

[11] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *Design, Automation & Test in Europe (DATE)*, 2011, pp. 1–6.

[12] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 25, no. 5, pp. 1694–1702, 2017.

[13] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of AIGs for error tolerant applications," in *International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2016, 83:1–83:8.

[14] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit carving: A methodology for the design of approximate hardware," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 545–550.

[15] L. Witschen, T. Wiersema, M. Artmann, and M. Platzner, "MUSCAT: MUS-based circuit approximation technique," in *Design, Automation & Test in Europe (DATE)*, IEEE, 2022, pp. 172–177.

[16] C.-T. Lee, Y.-T. Li, Y.-C. Chen, and C.-Y. Wang, "Approximate logic synthesis by genetic algorithm with an error rate guarantee," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2023, pp. 146–151.

[17] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate logic synthesis using boolean matrix factorization," in *Design Automation Conference (DAC)*, 2018, pp. 1–6.

[18] J. Ma, S. Hashemi, and S. Reda, "Approximate logic synthesis using boolean matrix factorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 1, pp. 15–28, 2021.

[19] C. Meng, W. Qian, and A. Mishchenko, "ALSRAC: Approximate logic synthesis by resubstitution with approximate care set," in *Design Automation Conference (DAC)*, 2020, 187:1–187:6.

[20] Y. Wu and W. Qian, "ALFANS: Multilevel approximate logic synthesis framework by approximate node simplification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 7, pp. 1470–1483, 2019.

[21] Z. Zhou, Y. Yao, S. Huang, S. Su, C. Meng, and W. Qian, "DALS: Delay-driven approximate logic synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, ACM, 2018, 86:1–86:7.

[22] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Automation & Test in Europe (DATE)*, 2013, pp. 1367–1372.

[23] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Design Automation Conference (DAC)*, IEEE, 2016, 128:1–128:6.

[24] G. Liu and Z. Zhang, "Statistically certified approximate logic synthesis," in *International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 344–351.

[25] K. S. Tam, C.-C. Lin, Y.-C. Chen, and C.-Y. Wang, "An efficient approximate node merging with an error rate guarantee," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 266–271.

[26] C. Meng, J. Sun, Y. Mai, and W. Qian, "MECALS: A maximum error checking technique for approximate logic synthesis," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[27] M. Rezaalipour, M. Biasion, I. Scarabottolo, G. A. Constantinides, and L. Pozzi, "A parametrizable template for approximate logic synthesis," in *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 175–178.

[28] S. Su *et al.*, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *Design Automation Conference (DAC)*, 2018, 54:1–54:6.

[29] S. Su, C. Meng, F. Yang, *et al.*, "VECBEE: A versatile efficiency-accuracy configurable batch error estimation method for greedy approximate logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[30] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, and L. Pozzi, "Partition and propagate: An error derivation algorithm for the design of approximate circuits," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[31] J. Echavarria, S. Wildermann, O. Keszocze, and J. Teich, "Probabilistic error propagation through approximated boolean networks," in *Design Automation Conference (DAC)*, 2020, pp. 1–6.

[32] X. Wang, S. Tao, J. Zhu, Y. Shi, and W. Qian, "AccALS: Accelerating approximate logic synthesis by selection of multiple local approximate changes," in *Design Automation Conference (DAC)*, 2023, pp. 1–6.

[33] C. Meng, Z. Zhou, Y. Yao, S. Huang, Y. Chen, and W. Qian, "HEDALS: Highly efficient delay-driven approximate logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[34] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *International Workshop on Logic Synthesis (IWLS)*, vol. 6, 2006, pp. 15–22.

[35] H. Riener *et al.*, "Scalable generic logic synthesis: One approach to rule them all," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[36] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 8, pp. 2573–2586, 2021.

[37] A. Mishchenko *et al.*, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 25, no. 5, pp. 743–755, 2006.

[38] P. McGeer, J. Sanghavi, R. Brayton, and A. S. Vincentelli, "Espresso-signature: A new exact minimizer for logic functions," in *Design Automation Conference (DAC)*, 1993, pp. 618–624.

[39] A. T. Calvino *et al.*, "A versatile mapping approach for technology mapping and graph optimization," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2022, pp. 410–416.

[40] A. Mishchenko *et al.*, *ABC: A system for sequential synthesis and verification*, http://people.eecs.berkeley.edu/~alanmi/abc/, 2024.

[41] Nangate, Inc., *Nangate 45nm open cell library*, https://si2.org/open-cell-library/, 2022.

[42] M. Hansen *et al.*, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design and Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[43] Brown University Scale Lab, *BACS: Benchmarks for approximate circuit synthesis*, https://github.com/scale-lab/BACS, 2024.

[44] EPFL Integrated Systems Laboratory, *The EPFL combinational benchmark suite*, https://lsi.epfl.ch/page-102566-en-html/benchmarks/, 2024.

[45] X. Wang, Z. Yan, C. Meng, Y. Shi, and W. Qian, "DASALS: Differentiable architecture search-driven approximate logic synthesis," in *International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[46] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer, 1991.

Chang Meng is a postdoctoral researcher at the Integrated Systems Laboratory, EPFL Lausanne, Switzerland. He received his Ph.D. degree in Electronic Science and Technology at Shanghai Jiao Tong University in 2023. His research interest is electronic design automation for emerging computing paradigms, especially the logic synthesis and verification of approximate computing circuits. His research work was nominated for the Best Paper Award at Design, Automation, and Test in Europe Conference (DATE).

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993 and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997. In 2002, he joined the EECS Department, University of California at Berkeley, Berkeley, CA, USA, where he is currently a Full Researcher. His current research interests include computationally efficient logic synthesis and formal verification.

Weikang Qian is an associate professor in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. He received his Ph.D. degree in Electrical Engineering at the University of Minnesota in 2011 and his B.Eng. degree in Automation at Tsinghua University in 2006. His main research interests include electronic design automation and digital design for emerging computing paradigms. His research works were nominated for the Best Paper Awards at International Conference on Computer-Aided Design (ICCAD), Design, Automation, and Test in Europe Conference (DATE), and International Workshop on Logic and Synthesis (IWLS). He serves as an associate editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. He is a senior member of IEEE.

Giovanni De Micheli is Professor and Director of the Integrated Systems Laboratory at EPFL Lausanne, Switzerland. Previously, he was Professor of Electrical Engineering at Stanford University. He holds a Nuclear Engineer degree (Politecnico di Milano, 1979), a M.S. and a Ph.D. degree in Electrical Engineering and Computer Science (University of California at Berkeley, 1980 and 1983).

He is a Fellow of ACM, AAAS and IEEE, a member of the Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies. He is also interested in heterogeneous platform design including electrical components and biosensors, as well as in data processing of biomedical information. He is member of the Scientific Advisory Board of IMEC (Leuven, B) and STMicroelectronics.

Professor De Micheli is the recipient of the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards.