Simulation-Guided Approximate Logic Synthesis Under the Maximum Error Constraint

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

Abstract—**Approximate computing is an effective computing paradigm for improving the energy efficiency of error-tolerant applications. Approximate logic synthesis (ALS) is an automatic process to generate approximate circuits with reduced area, delay, and power, while satisfying user-specified error constraints. This paper focuses on ALS under the maximum error constraint. As an essential error metric that provides a worst-case error guarantee, the maximum error is crucial for many applications such as image processing and machine learning. This work proposes an efficient simulation-guided ALS flow that handles this constraint. It utilizes logic simulation to 1) prune local approximate changes (*LACs*) with large errors that violate the error constraint, and 2) accelerate the SAT-based LAC selection process. Furthermore, to enhance scalability, our ALS flow iteratively selects a set of promising LACs satisfying the error constraint to improve efficiency. The experimental results show that compared with the state-of-the-art method, our ALS flow accelerates by** $30.6\times$**, and further reduces the circuit area and delay by** 18.2% **and** 4.9%**, respectively. Notably, our flow scales to large EPFL benchmarks with up to 38540 nodes, which remain challenging for existing ALS methods tackling maximum error constraint.**

Index Terms—**approximate logic synthesis, approximate computing, maximum error, logic simulation**

I. INTRODUCTION

Approximate computing [1] is an emerging low-power design paradigm for error-tolerant applications such as signal processing, data mining, and machine learning [2]. It carefully introduces errors to significantly reduce the hardware cost, while the application-level quality is almost unaffected. *Approximate logic synthesis* (*ALS*) is an automatic process to generate approximate circuits [3]. An ALS tool takes an accurate circuit and user-specified error constraints as inputs and outputs an approximate circuit with smaller area, delay, and power, satisfying the constraints.

To evaluate the accuracy of an approximate circuit, two types of error metrics are utilized, the average error and maximum error [3]. Average error, such as error rate and mean error distance, measures the average deviation between the outputs of the accurate and approximate circuits, while the maximum error measures the maximum deviation between the outputs of the accurate and approximate circuits over all input patterns. Typical maximum errors include *maximum error distance* (*MaxED*) and *maximum Hamming distance* (*MaxHD*). Maximum error provides a worst-case guarantee of the error, which is crucial for many

applications. For instance, in image processing, even if the average error is low, occasional large errors can lead to visible artifacts. Another example is machine learning, where rare extreme deviations may cause mispredictions. Using maximum error constraints ensures that every computation remains within safe bounds, preserving overall quality and reliability. Considering the importance of maximum error, this paper focuses on the ALS under the maximum error constraint.

Many ALS methods for maximum error constraint have been proposed [4]–[7], most of which simplify the circuit by applying *local approximate changes* (*LACs*). A LAC is a local modification of the circuit. For example, a *constant LAC* [8] replaces a signal by a constant 0 or 1, and a *SASIMI LAC* [9] substitutes a signal by another. After generating candidate LACs, an ALS flow estimates the maximum error caused by the LACs. Based on the error estimation results, the ALS flow then identifies LACs that can be applied to simplify the circuit while satisfying the error constraint. There are two categories of maximum error estimation methods. The first category estimates an upper bound of the maximum error [5], [10]–[12]. However, these methods only support the simple constant LAC [8], while complex LACs such as the SASIMI LAC [9] that can achieve better approximate circuits are not supported. To handle complex LACs, the second category checks whether the maximum error is within a user-specified bound or not [13]–[15]. For each LAC, these methods convert the maximum error checking problem for the LAC into a SAT problem, and the SAT solving result determines whether the maximum error caused by the LAC is within the bound or not. However, a new challenge is the massive number of complex LACs in a circuit. The solving of their corresponding numerous SAT problems is time-consuming and limits the scalability of the ALS flow. For example, for a circuit with N nodes, there are $O(N^2)$ SASIMI LACs [9] that replace a signal by another, and solving the corresponding $O(N^2)$ SAT problems is impractical for large circuits.

To address the above challenges, we propose an efficient logic simulation-guided ALS flow under the maximum error constraint, which can handle complex LACs and scale to large circuits. Logic simulation has shown its effectiveness in accelerating the SAT solving process in many traditional logic synthesis works such as [16], and this motivates us to leverage it to accelerate the maximum error checking process in ALS. Our main contributions are as follows:

1) We propose to utilize logic simulation to prune large-error LACs violating the maximum error constraint, which significantly reduces the number of LACs to be considered, and hence accelerates the ALS flow.
2) For the remaining LACs after pruning, we propose to

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

Weikang Qian is with the Global College, 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: Chang Meng.

use simulation-guided SAT solving to further accelerate the LAC selection process.

3) Based on the simulation-based LAC pruning and fast LAC selection, we design an ALS flow for maximum error constraint that iteratively applies a set of promising LACs for efficient circuit simplification.

The experimental results show that compared with the state-of-the-art method, our ALS flow accelerates by $30.6\times$, and further reduces the circuit area and delay by 18.2% and 4.9%, respectively. Our method scales to large EPFL benchmarks with up to 38540 nodes, which remain challenging for existing ALS methods tackling the maximum error constraint. Our work is open-source and available at https://github.com/changmg/SimALS-MaxError.

The remainder of this paper is organized as follows. Section II reviews the related works. Section III introduces the background. Section IV elaborates the proposed simulation-guided ALS flow under the maximum error constraint. The experimental results are presented in Section V, followed by conclusions in Section VI.

II. RELATED WORKS

This section reviews the related works on ALS methods and maximum error estimation methods for ALS.

A. Approximate Logic Synthesis (ALS) Methods

As mentioned before, the average error and the maximum error are two typical error metrics used in ALS. Some ALS methods can handle both average and maximum error constraints, such as [8] and [14], some focus on the average error constraint, such as [9], [17]–[22], and others deal with the maximum error constraint, such as [4]–[7]. Our work differs from the above works on ALS for maximum error constraint in two aspects: 1) supporting complex LACs for better circuit quality, and 2) using effective simulation-guided techniques to enhance scalability.

B. Error Estimation Methods for ALS

Error estimation of LACs is a critical step in ALS. The average error estimation is usually based on Monte Carlo simulation [23]–[25] or analytical methods [26]. However, these methods cannot be directly applied for maximum error estimation.

To estimate the maximum error caused by LACs, two categories of methods are proposed. The first category estimates a maximum error upper bound, such as [10], [12]. However, both [10] and [12] only consider a simple LAC that replaces a signal by a constant 0 or 1 [8]. Complex LACs such as the SASIMI LAC [9] that can further simplify the circuit are not supported.

The second category directly checks whether the maximum error caused by each LAC is within the error bound, such as [6], [13]–[15], [27]. For example, the MUSCAT ALS flow [6] encodes the maximum error checking of all candidate LACs into a single *minimal unsatisfiable subset* (*MUS*) problem. The solution of the MUS problem corresponds to an optimized approximate circuit satisfying the maximum error constraint. However, solving the MUS problem is very time-consuming, and MUSCAT does not support complex LACs like the SASIMI LAC [9]. Although

MUSCAT sets a time limit for the MUS solving to enhance scalability, the time limit may lead to suboptimal approximate circuits. Another approach, the MECALS ALS flow [27], converts the maximum error checking for all candidate LACs into a SAT sweeping problem. Unfortunately, SAT sweeping also has a scalability issue, making MECALS unable to handle large circuits in practice. Our method, instead, uses logic simulation to accelerate the check of maximum errors caused by numerous LACs, which belongs to the second category of maximum error estimation methods. Unlike the existing works, our method can handle complex LACs and has better scalability.

III. 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 *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 sequence of connected nodes in the circuit. If there exists a path from node u to v, then v is a *transitive fanout* (*TFO*) of u.

B. Maximum Error Metrics

Consider two multiple-output Boolean functions $\boldsymbol{y}$: $\mathbb{B}^I \to \mathbb{B}^O$ for an accurate circuit G and $\hat{\boldsymbol{y}}$: $\mathbb{B}^I \to \mathbb{B}^O$ for its approximate counterpart $\hat{G}$, where I and O are the numbers of PIs and POs, respectively. The maximum error of circuit $\hat{G}$ quantifies the maximum deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ over all PI patterns $\boldsymbol{x}$ as follows:

$$MaxError(G) = \max_{\boldsymbol{x} \in \mathbb{B}^I} D\left(\boldsymbol{y}(\boldsymbol{x}), \hat{\boldsymbol{y}}(\boldsymbol{x})\right), \qquad (1)$$

where $\boldsymbol{y}(\boldsymbol{x})$ and $\hat{\boldsymbol{y}}(\boldsymbol{x})$ are binary vectors of length O, denoting the PO values of G and $\hat{G}$ under the PI pattern $\boldsymbol{x}$, respectively. The function D is called a deviation function, measuring the deviation between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$.

Typical maximum errors include *maximum error distance* (*MaxED*) and *maximum Hamming distance* (*MaxHD*). MaxED measures the maximum absolute difference between the numerical values encoded by the POs of the accurate and approximate circuits. Its deviation function is

$$D_{\text{MaxED}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = |int(\boldsymbol{y}) - int(\hat{\boldsymbol{y}})|, \qquad (2)$$

where $int(\boldsymbol{v})$ returns the integer encoded by the binary vector $\boldsymbol{v}$. For example, if $\boldsymbol{y}$ encodes an O-bit unsigned integer, then $int(\boldsymbol{y}) = \sum_{k=1}^{O} 2^{k-1} y_k$, where y_k denotes the k-th bit of the binary vector $\boldsymbol{y}$. By measuring the numerical deviation, MaxED is a suitable metric for arithmetic circuits, such as adders and multipliers.

MaxHD measures the maximum number of bit-flips between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. Its deviation function is

$$D_{MaxHD}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{k=1}^{O} |y_k - \hat{y}_k|. \qquad (3)$$

By limiting the number of bit-flips, MaxHD is a suitable metric for digital communication and error correction circuits.

C. Error Miter for Maximum Error Checking

An error miter is an auxiliary circuit to check whether the maximum error of an approximate circuit exceeds a given bound or not [13], [28]. As shown in Fig. 1, it consists of an accurate circuit G, an approximate circuit $\hat{G}$, a deviation function unit, and a comparator. The accurate and approximate circuits take the same PIs x as inputs, and their corresponding outputs are y and $\hat{y}$, respectively. The deviation unit computes the deviation function $D(y, \hat{y})$, such as Eq. (2) or Eq. (3). The comparator checks whether $D(y, \hat{y})$ is larger than the bound B. If $D(y, \hat{y}) > B$, the output of the comparator, f, is 1; otherwise, f is 0.

To check the maximum error of circuit $\hat{G}$, the error miter is converted into a SAT problem. If the solver returns SAT, then there exists a PI pattern x causing $f = 1$. In this case, we have $D(y, \hat{y}) > B$, and hence $MaxError(\hat{G}) > B$. Otherwise, if the solution is UNSAT, then f is always 0. This means that $D(y, \hat{y}) \leq B$ over all PI patterns x, implying $MaxError(\hat{G}) \leq B$.
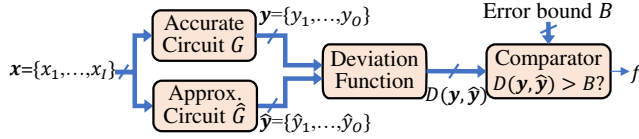

Fig. 1. An error miter that checks whether the maximum error of the approximate circuit $\hat{G}$ exceeds the error bound B or not.

IV. METHODOLOGY

This section introduces our simulation-guided ALS flow under the maximum error constraint. We first overview the flow in Section IV-A, followed by the details in Sections IV-B and IV-C.

A. Overview

Our ALS flow aims to solve this problem: Given an accurate circuit G in any graph representation (*e.g.*, AIG, gate netlist, etc.) and a maximum error bound B, find a min-area approximate gate netlist G_{final}, while ensuring $MaxError(G_{final}) \leq B$.

As shown in Fig. 2, our flow starts by initializing a *current approximate circuit* $\hat{G}$ as a copy of the accurate circuit G. Then, circuit $\hat{G}$ is iteratively simplified in a main loop, indicated by the blue arrows in Fig. 2. Each iteration consists of three key steps. Step 1 generates a set of candidate LACs L_{cand}. Here, the generated LACs can be any *single-output LACs*, *i.e.*, LACs whose affected local circuits have only one output, such as constant LACs and SASIMI LACs. Since L_{cand} usually contains numerous LACs, step 2 prunes the invalid LACs violating the error constraint according to logic simulation results. The set of remaining LACs after pruning is denoted as L_{rem}. Then, step 3 selects a set of promising LACs from L_{rem} and applies them to simplify the current approximate circuit $\hat{G}$, where a promising LAC refers to a LAC whose application significantly reduces circuit area while satisfying the error constraint. If the approximate circuit $\hat{G}$ is successfully simplified compared to the previous iteration, the main loop continues for the next iteration. Otherwise, no more valid LACs exist, and the main loop terminates. Then, traditional logic synthesis is performed to further simplify the circuit

$\hat{G}$ without introducing additional errors, producing the final approximate gate netlist G_{final}.

Note that logic simulation serves as a guider in our ALS flow. As shown in the middle right part of Fig. 2, the circuit simulator not only guides the pruning of invalid LACs in L_{cand}, but also accelerates the LAC selection process by guiding the SAT solving. Furthermore, the simulation patterns in the circuit simulator are updated on the fly by the LAC selection information from step 3. This technique further accelerates the ALS flow by reducing the number of SAT problems to be solved.

The following subsections detail the key steps in our ALS flow. Specifically, Section IV-B introduces step 2, the simulation-guided LAC pruning, and Section IV-C describes step 3, the selection and application of the promising LACs using simulation-guided SAT solving.
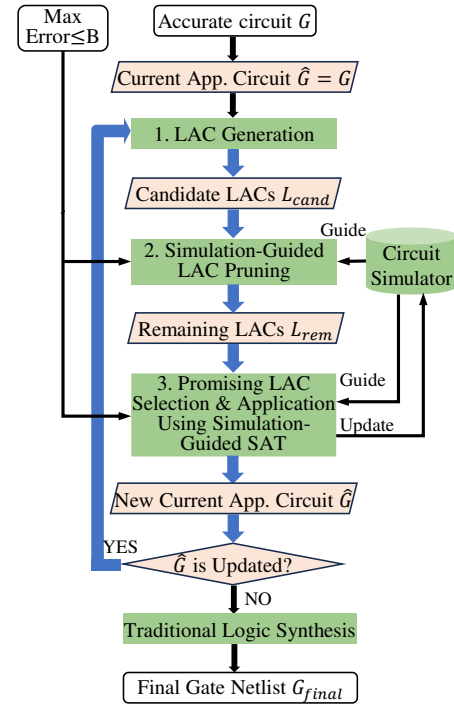

Fig. 2. Simulation-guided ALS flow under the maximum error constraint.

B. Simulation-Guided LAC Pruning

The LAC pruning step filters out some invalid candidate LACs in L_{cand} generated in step 1 of our flow and returns a set of remaining LACs, denoted as L_{rem}. There are usually many LACs in L_{cand}. For example, a circuit with N nodes has $O(N^2)$ SASIMI LACs [9] that replace a node by another. If we use the error miter-based method (see Section III-C) to check the maximum error of each candidate LAC in L_{cand}, then $O(N^2)$ SAT problems need to be solved, which is impractical for large circuits. Given that simulating a small subset of all possible input patterns (*e.g.*, $\leq 2^{13}$ patterns in our experiments) is typically much faster than SAT solving, we propose to use simulation to quickly prune the invalid LACs in L_{cand} violating the maximum error constraint. After the pruning, the number of LACs to be checked by SAT solving is significantly reduced, thus improving the efficiency of the ALS flow.

In the following parts, we first introduce a theoretical foundation of the simulation-guided LAC pruning in Section IV-B1, *i.e.*, simulation can obtain a lower bound of the maximum error caused by a candidate LAC. Next, we

describe how to efficiently compute the lower bound on the maximum error in Section IV-B2.

1) Theoretical Foundation: Given a candidate LAC l_{cand}, logic simulation over a subset of PI patterns yields the maximum observable error under these patterns, denoted as $MaxError_{LB}(l_{cand})$. Since the simulation does not cover all PI patterns, this value is a lower bound on the real maximum error caused by l_{cand}, *i.e.*,

$$MaxError_{LB}(l_{cand}) \leq MaxError_{real}(l_{cand}). \tag{4}$$

This property forms the basis for the simulation-guided LAC pruning. If simulation finds that $MaxError_{LB}(l_{cand}) > B$, then l_{cand} is guaranteed to violate the maximum error constraint and is discarded. In our implementation, we first simulate with a small number of M_{small} random PI patterns to roughly prune the invalid LACs, followed by a larger number of M PI patterns to obtain a tighter lower bound of the maximum error for further pruning. The LACs not pruned by the two-round simulation are retained in L_{rem}.

2) Efficient Computation of the Maximum Error Lower Bounds: The theoretical foundation of the simulation-guided LAC pruning in Eq. (4) requires computing $MaxError_{LB}(l_{cand})$ for each candidate LAC l_{cand}. To compute $MaxError_{LB}(l_{cand})$ for each candidate LAC l_{cand}, a naive way is to apply l_{cand} to the current approximate circuit $\hat{G}$ and obtain a new approximate circuit $\hat{G}_{cand}$. After simulating the accurate circuit G and the approximate circuit $\hat{G}_{cand}$ under some sampled PI patterns, $MaxError_{LB}(l_{cand})$ can be obtained. This method is straightforward but slow, requiring $O(|L_{cand}|)$ simulation runs, where $|L_{cand}|$ is the number of candidate LACs.

Instead of using the naive method, we accelerate the computation of all $MaxError_{LB}(l_{cand})$'s based on the change propagation matrix (CPM) proposed in [23]. The CPM P for the current approximate circuit $\hat{G}$ is a three-dimensional 0-1 matrix of size $M \times N \times O$, where M is the number of simulation patterns, N is the number of functional nodes in the circuit, and O is the number of POs in the circuit. Each entry in the CPM is indexed as $P[i, n, \hat{y}_k]$, where $1 \leq i \leq M$ represents the i-th simulation pattern, n is a functional node in the circuit, and $\hat{y}_k$ $(1 \leq k \leq O)$ is the k-th PO in $\hat{G}$. The entry $P[i, n, \hat{y}_k]$ evaluates the impact of the change in n's value on $\hat{y}_k$. Specifically, $P[i, n, \hat{y}_k] = 1$ indicates that a flip of n's value will cause a flip of y_k's value under the i-th pattern, while $P[i, n, \hat{y}_k] = 0$ means that $\hat{y}_k$'s value keeps unchanged after a flip of n's value under the i-th pattern. To compute $P[i, n, \hat{y}_k]$, under the i-th pattern we can flip n's value, update the values of all n's TFOs and POs using n's new value, and then check whether $\hat{y}_k$'s value changes. If it changes, then $P[i, n, \hat{y}_k]$ is 1; otherwise, $P[i, n, \hat{y}_k]$ is 0. We apply the above process to each functional node n in the circuit to obtain its CPM entries. Thus, computing the CPM for all functional nodes requires $O(N)$ simulation runs, where N is the number of functional nodes.

Example 1 *In the example circuit shown in Fig. 3, assume that the i-th PI pattern for simulation is $x_1 x_2 \ldots x_5 = 11101$. The simulation values of the gates are shown above the wires. To compute $P[i, n, \hat{y}_0]$ and $P[i, n, \hat{y}_1]$, we flip node n's value from 0 to 1, and update the values of n's TFOs. That is, the value of b changes from 0 to 1, the value of c changes from 0 to 1, the values of d and $\hat{y}_0$ change*
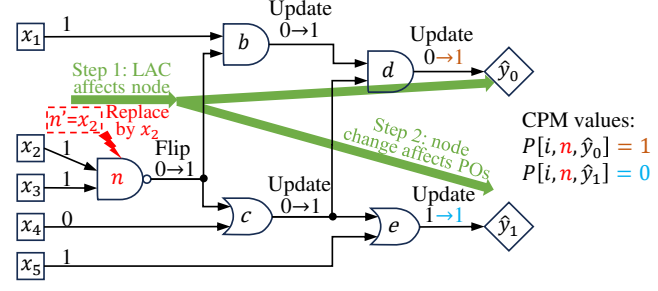

Fig. 3. An example circuit. The number above each wire is the signal value under the i-th input pattern in the simulation. The impact of the LAC that replaces node n with node x_2 is considered.

from 0 to 1, and the values of e and $\hat{y}_1$ keep unchanged. Since under the i-th pattern, the flip of n's value causes a flip of $\hat{y}_0$'s value, while it does not affect $\hat{y}_1$'s value, we have $P[i, n, \hat{y}_0] = 1$ and $P[i, n, \hat{y}_1] = 0$.

CPM can be used to efficiently compute the PO values after applying each candidate LAC, and hence the lower bound on the maximum error of the candidate LAC. For the single-output LACs considered in this work, each of them, denoted as l_{cand}, can be modeled as replacing an existing node n in the circuit with a new node n'. After applying l_{cand} to the current approximate circuit $\hat{G}$, we can obtain the new value of the k-th PO under the i-th pattern, denoted as $y_k'[i]$, as follows:[1]

$$\begin{aligned} y_k'[i] &= \hat{y}_k[i] \oplus Impact[i, l_{cand}, \hat{y}_k] \\ &= \hat{y}_k[i] \oplus ((n[i] \oplus n'[i]) \wedge P[i, n, \hat{y}_k]), \end{aligned} \tag{5}$$

where $\hat{y}_k[i]$ is the k-th PO's value under the i-th pattern before applying l_{cand}, $Impact[i, l_{cand}, \hat{y}_k]$ is a binary value called the *impact factor* for evaluating the impact of applying l_{cand} on the k-th PO under the i-th pattern, which will be explained next, $n[i]$ and $n'[i]$ are the values of node n and n' under the i-th pattern, respectively, and $\oplus$ and $\wedge$ are the XOR and AND operations, respectively.

Note that Eq. (5) has an impact factor $Impact[i, l_{cand}, \hat{y}_k]$, measuring whether applying l_{cand} changes the value of k-th PO $(\hat{y}_k)$ under the i-th pattern. The impact factor has two components, corresponding to the two steps of the impact of l_{cand} on the PO, indicated by the green arrows in Fig. 3. Step 1 is that the application of l_{cand} changes node n's value under the i-th pattern, which is captured by $n[i] \oplus n'[i]$. If the value of the new node n' is different from that of the original node n under the i-th pattern, then $n[i] \oplus n'[i]$ is 1, indicating the change of n's value after applying l_{cand}. Otherwise, $n[i] \oplus n'[i]$ is 0, indicating that applying l_{cand} does not change node n's value, and hence does not affect $\hat{y}_k$'s value under the i-th pattern. Step 2 is that the change of n's value causes the change of $\hat{y}_k$'s value under the i-th pattern, which is captured by the CPM entry $P[i, n, \hat{y}_k]$. It is obvious that only if the two steps both occur, $\hat{y}_k$'s value changes after applying l_{cand} under the i-th pattern. Therefore, the impact factor $Impact[i, l_{cand}]$ is computed by the AND of the two components, *i.e.*, $(n[i] \oplus n'[i]) \wedge P[i, n, \hat{y}_k]$, as shown in Eq. (5). Finally, the new PO value $y_k'[i]$ is obtained by XORing the original PO value $\hat{y}_k[i]$ with the binary impact factor $Impact[i, l_{cand}]$.

---

[1]Eq. (5) is an intermediate result derived from [23]. However, [23] studies the ALS problem under the average error constraint. Here, we extend its use to the ALS problem under the maximum error constraint, where it serves as a key building block of our simulation-guided ALS flow.

For each candidate LAC l_{cand}, after obtaining the new PO values $y'_k[i]$'s for all POs under all simulation patterns, we can further obtain the deviation between the PO values of the accurate circuit G and those of the approximate circuit after applying l_{cand}. Then, $MaxError_{LB}(l_{cand})$ can be obtained as the maximum deviation over all simulation patterns.

Example 2 *For the example circuit in Fig. 3, consider a LAC l_{cand} that replaces node n with another node x_2. Before applying l_{cand}, the values of $\hat{y}_0$ and $\hat{y}_1$ under the i-th simulation pattern, denoted as $\hat{y}_0[i]$ and $\hat{y}_1[i]$, are 0 and 1, respectively. From Example 1, we know $P[i, n, \hat{y}_0] = 1$ and $P[i, n, \hat{y}_1] = 0$. Then, the impact factors can be computed as follows:*

$$Impact[i, l_{cand}, \hat{y}_0] = (n[i] \oplus x_2[i]) \wedge P[i, n, \hat{y}_0] = 1,$$

$$Impact[i, l_{cand}, \hat{y}_1] = (n[i] \oplus x_2[i]) \wedge P[i, n, \hat{y}_1] = 0.$$

This means that applying l_{cand} changes $\hat{y}_0$'s value under the i-th pattern, while it does not affect $\hat{y}_1$'s value. Therefore, after applying l_{cand}, the new values of $\hat{y}_0$ and $\hat{y}_1$ under the i-th simulation pattern, denoted as $y'_0[i]$ and $y'_1[i]$, can be updated as follows:

$$y'_0[i] = \hat{y}_0[i] \oplus Impact[i, l_{cand}, \hat{y}_0] = 0 \oplus 1 = 1,$$

$$y'_1[i] = \hat{y}_1[i] \oplus Impact[i, l_{cand}, \hat{y}_1] = 1 \oplus 0 = 1.$$

If we consider the MaxED metric and assume that the PO values of the accurate circuit G under the i-th simulation pattern are $y_0[i] = 0$ and $y_1[i] = 1$, then the deviation between the PO values of G and those of the approximate circuit after applying l_{cand} under the i-th simulation pattern is

$$D(\boldsymbol{y}[i], \hat{\boldsymbol{y}}[i]) = |(2y_1[i] + y_0[i]) - (2y'_1[i] + y'_0[i])| = 1.$$

After computing the deviation $D(\boldsymbol{y}[i], \hat{\boldsymbol{y}}[i])$ for all simulation patterns, the maximum value of the deviation is $MaxError_{LB}(l_{cand})$, i.e., the lower bound on the maximum error caused by the LAC l_{cand}.

Using the CPM-based method to compute the lower bounds on the maximum errors of all candidate LACs, the main computational effort lies in constructing the CPM. As mentioned above, obtaining the CPM for the current approximate circuit $\hat{G}$ requires only $O(N)$ simulation runs, where N is the number of functional nodes in $\hat{G}$. Compared with the naive method with $O(|L_{cand}|)$ simulation runs, the CPM-based method is much more efficient, as N is usually much smaller than $|L_{cand}|$. After obtaining the lower bounds on the maximum errors of all candidate LACs, the LAC pruning step can efficiently filter out the invalid LACs that violate the maximum error constraint based on Eq. (4) and return the set of remaining LACs L_{rem} to be further checked by the promising LAC selection and application step.

C. Promising LAC Selection and Application Based on Simulation-Guided SAT Solving

As shown in Fig. 2, the promising LAC selection and application step is responsible for checking the validity of the remaining LACs in L_{rem} after the simulation-based pruning, selecting a subset of promising LACs L_{prom} to reduce the circuit area as much as possible, and applying all LACs in L_{prom} to simplify the current approximate circuit $\hat{G}$.

In the following parts, we will first formulate the LAC selection problem in Section IV-C1 and then introduce a greedy LAC selection strategy supported by simulation-guided SAT solving in Section IV-C2. Finally, we will discuss the order of checking LACs in Section IV-C3, which significantly affects the quality of the final approximate circuit and the efficiency of the promising LAC selection and application step.

1) Formulation of the LAC Selection Problem: The LAC selection problem can be formulated as: Given a current approximate circuit $\hat{G}$ and a set of LACs L_{rem}, find a subset of promising LACs $L_{prom} \subseteq L_{rem}$ to reduce the circuit area as much as possible, while satisfying the following three constraints:

- Error constraint: the maximum error caused by applying all LACs in L_{prom} is within the error bound B.
- Circuit integrity constraint: the LACs in L_{prom} do not introduce a logic loop in the circuit.
- LAC conflict constraint: at most one LAC can be applied to each node in the circuit, since two LACs cannot be applied to the same node simultaneously.

Note that our framework supports circuits represented in any graph format, such as AIGs and gate netlists. For AIGs, circuit area is estimated by counting the number of AND nodes, while for gate netlists, it is obtained by summing the areas of all gates.

Our LAC selection problem is similar to the one in the MUSCAT method [6], which is a state-of-the-art ALS method under the maximum error constraint. However, the formulation of MUSCAT only considers the constant LACs that replace a signal by a constant 0 or 1, while ours considers arbitrary single-output LACs. To solve this NP-hard combinatorial optimization problem, MUSCAT converts the problem into a MUS problem and solves it using MUS solvers, which is time-consuming and limits its scalability. Although MUSCAT sets a time limit for the MUS solving to enhance the scalability, the time limit leads to suboptimal approximate circuits. In our work, we propose a greedy selection strategy to obtain a good solution efficiently, which will be introduced next.

2) Greedy LAC Selection Strategy Supported by Simulation-Guided SAT Solving: As shown in Fig. 4, our method first sorts the LACs in L_{rem} and keeps the top K, where K is a user-defined parameter. More details about the sorting are discussed in Section IV-C3. Then, each LAC in the sorted list is examined in order to determine whether it should be applied. Denote the initial circuit before applying any LAC as $G_0 = \hat{G}$ (the current approximate circuit) and the updated circuit after processing the j-th ($1 \leq j \leq K$) LAC as G_j. After processing all K LACs, the resulting circuit G_K becomes the updated current approximate circuit $\hat{G}$ for subsequent iterations. Note that the j-th candidate LAC l_j affects the circuit G_{j-1}. When we process LAC l_j, we first check whether applying it to the circuit G_{j-1} will introduce a logic loop in the circuit. If so, we skip l_j and keep the circuit G_j the same as G_{j-1}. Then, for the LAC l_j that does not introduce a logic loop, we check its validity using the error miter-based method introduced in Section III-C. Specifically, we build an error miter (see Fig. 1) using the accurate circuit G and the approximate circuit after applying l_j to G_{j-1}, convert

the miter into a SAT problem, and then use a SAT solver to solve the problem. The solution of the SAT problem determines whether to apply l_j or not. There are three possible results of the SAT solving for l_j:

- UNSAT: This indicates that the maximum error caused by LAC l_j is no larger than B. In this case, LAC l_j is valid, and we apply it to the circuit G_{j-1}, obtaining the resulting circuit G_j. Examples of this case are l_1 and l_3 in Fig. 4. Note that in order to satisfy the LAC conflict constraint, after applying l_j, we remove all LACs that affect the same node as l_j from the top K LACs.

- SAT: This indicates that applying LAC l_j to the circuit G_{j-1} causes a maximum error that exceeds B. In this case, LAC l_j is invalid and should be skipped, and circuit G_j keeps the same as circuit G_{j-1}. Examples of this case are l_2 and l_K in Fig. 4.

- UNDEFINED: This happens when the SAT solver cannot give a solution within a computing resource limit. In this case, the solver cannot determine whether l_j is valid or not. To avoid violating the error constraint, conservatively, we do not select l_j and skip it, and circuit G_j keeps the same as circuit G_{j-1}. In our implementation, we set a maximum conflict number for the SAT solver to avoid the long runtime of SAT solving. In a SAT solver, a conflict happens when the current variable assignments make a clause false. During SAT solving, if the number of conflicts exceeds the maximum conflict number, the solver returns UNDEFINED. Moreover, if the SAT solver returns UNDEFINED for LAC l_j, for efficiency, we add l_j into a blacklist and do not consider it again in future iterations of the ALS flow. An example of this case is l_4 in Fig. 4.
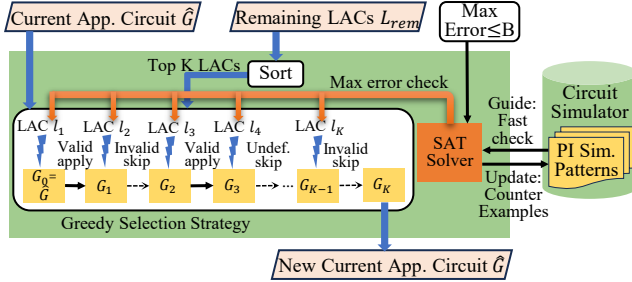

Fig. 4. Greedy promising LAC selection strategy supported by simulation-guided SAT solving.

When the solver returns SAT for LAC l_j, a counter-example pattern is also returned, which is a PI pattern that causes the maximum error to exceed B. Note that the counter-example pattern is not sampled in previous logic simulation in the LAC pruning step. It can be viewed as a sensitive pattern that activates the large deviation caused by LAC l_j. This sensitive pattern is very likely to activate large deviations caused by other LACs, especially those affecting the same node as l_j. Therefore, the counter-example pattern is very helpful in guiding the future maximum error checking of other LACs.

We propose a reuse mechanism of the counter-example patterns, as shown in the right part of Fig. 4. When examining LAC l_j, before solving the SAT problem for l_j, we perform logic simulation using the counter-example patterns generated by l_s's ($s < j$) to quickly check whether l_j already violates the maximum error bound under these patterns. If simulation finds that l_j is invalid, then we directly skip the LAC without solving the SAT problem. Otherwise, we still need to solve the SAT problem for l_j to check its validity. By guiding the SAT solving with the simulation using the counter-example patterns, we can skip unnecessary SAT solving and improve the efficiency of the LAC selection and application step. Furthermore, we also update the simulation patterns by storing the counter-example patterns. In this way, the stored counter-example patterns will guide the future iterations of the ALS flow in their simulation-based LAC pruning and promising LAC selection and application steps. In practice, this method reduces the number of SAT problems to be solved in the ALS flow and accelerates the ALS flow.

3) The Order of Checking and Selecting LACs: As shown in Fig. 4, before applying the greedy LAC selection strategy, an essential step is to sort the LACs in L_{rem}. After sorting, the top K LACs are kept and the other LACs are discarded.

The sorting of LACs is crucial for both the quality of the final approximate circuit and the efficiency of the promising LAC selection and application step. Note that in the greedy LAC selection strategy, a previously selected LAC changes the circuit structure, and hence affects the validity of the following LACs. For example, in Fig. 4, l_1 is selected and applied to the circuit, affecting the validity of $l_2, l_3, \ldots, l_K$. Thus, different orders of LACs may lead to different results of the LAC selection and finally affect the quality of the final approximate circuit. Moreover, a poor order of LACs may slow down the LAC selection process. For example, if invalid LACs are checked first, then the promising LAC selection and application step may spend a lot of time checking and discarding them with no simplification of the circuit, hence wasting time.

We propose a sorting strategy based on two criteria: 1) the maximum error lower bound (primary) and 2) the estimated area reduction of the LAC (secondary). First, the LACs are sorted in the ascending order of their maximum error lower bounds, obtained from logic simulation. The motivation is that only the top K LACs are kept after the sorting. We want to keep as many valid LACs as possible within the top K so that the circuit area can be effectively reduced by applying these valid LACs. Given that a LAC with a small maximum error lower bound is more likely to be valid, sorting the LACs by the lower bounds can increase the probability of keeping valid LACs in the top K LACs.

Only if multiple LACs share the same maximum error lower bound, they are further sorted in descending order of the estimated area reduction, prioritizing LACs that reduce more area. Our method supports circuits represented in any graph format, such as AIGs and gate netlists. For AIGs, a LAC's area reduction is estimated by counting the nodes in the maximum fanout-free cone (MFFC) [29] of the node where the LAC is applied, instead of performing time-consuming logic synthesis to obtain the exact reduction. For gate netlists, a LAC's area reduction is obtained by summing the areas of the gates in the corresponding MFFC.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2026.3654911

7

V. Experimental Results

A. Experimental Setup

We implement the proposed ALS flow under the maximum error constraint in C++ and test it on a single core of an AMD Ryzen 9 5900X processor with 64GB RAM. Our flow is developed upon a state-of-the-art open-source logic synthesis and verification system, ABC [30]. The flow also integrates CryptoMiniSat [31], a modern SAT solver with rich features and a simple interface. To avoid long runtime of SAT solving, we set a maximum conflict number of 2^{18} for each SAT problem.

In all experiments, the initial circuits are represented as AIGs, although our flow can also handle other circuit representations, such as gate netlists. The reason for using AIGs is that many AIG-based logic synthesis works [32], [33] and ALS works [20], [34] have shown significant advantages in reducing hardware cost, particularly for CMOS technologies. The standard cell library used in our experiments is the Nangate 45nm library [35]. The traditional logic synthesis in the last step of our flow (see Fig. 2) consists of AIG optimization and technology mapping. For AIG optimization, we apply the ABC script *"resyn2rs"* for 3 iterations. This script combines four transformations, *i.e.*, *rewriting*, *refactoring*, *balancing*, and *resubstitution*, to simplify the local structure of the AIG, and has been shown effective and widely used in recent works [36]–[39]. Iteratively applying the script can further reduce the AIG size and depth, and in our experiments, 3 iterations provide a good trade-off between circuit quality and runtime. For technology mapping, we use the standard area-oriented mapping script *"dch; amap"* to convert the AIG into a gate netlist, as recommended in the ABC tutorial [40]. Unless otherwise specified, the following default parameters are used in all experiments. For the simulation-based LAC pruning (see Section IV-B1), we first use $M_{small} = 2^{10}$ simulation patterns to quickly filter out LACs inducing large errors, and then use $M = 2^{13}$ simulation patterns for a more fine-grained filtering. For the parameter K in the sorting strategy in Section IV-C3, we set $K = 100$. To build the error miter in Fig. 1 for maximum error checking, we use Verilog to describe the error miter and then use Yosys [41] and ABC to synthesize the Verilog description.

To evaluate the hardware cost, we measure the area, delay, and power consumption of the synthesized gate netlist (post-synthesis, before place-and-route). The area is computed by summing the individual gate areas in the netlist. The delay is obtained using the static timing analysis command *"stime"* in ABC. The power is estimated with the Synopsys Design Compiler [42] at 2MHz under a uniform input distribution. We then define *area ratio*, *delay ratio*, and *power ratio* as the respective values of the approximate netlist normalized to those of the accurate one. Smaller ratios are preferred. To evaluate the accuracy of circuits, two different maximum error metrics, MaxED and MaxHD, are considered in our experiments. Note that the focus of this work is ALS under the maximum error constraint, so we do not compare it with other ALS methods under average error constraints, such as [34] and [43]. For all generated approximate circuits, the error miter in Fig. 1 is used to formally verify that the maximum errors of the circuits satisfy the given error bounds.

TABLE I
EXPERIMENTAL BENCHMARKS. AREA, DELAY, AND POWER ARE MEASURED BY MAPPING AIGS INTO GATE NETLISTS WITH THE NANGATE 45NM LIBRARY.

Benchmark suite	Circuit	#PIs/#POs	AIG		Gate netlist		
			Size	Depth	Area /um^2	Delay /ns	Power /μW
Used in MECALS [27]	absdiff	16/8	141	14	87.3	0.42	80.9
	add8	16/9	66	10	42.0	0.36	36.8
	add32	64/33	252	64	184.6	1.84	173.0
	binsqrd	16/18	1562	50	1052.3	1.53	989.5
	buttfly	32/34	265	48	170.5	1.01	172.6
	mac	12/8	145	20	92.8	0.60	73.4
	mult8	16/16	649	40	435.4	1.26	410.2
	mult16	32/32	1981	72	1418.8	1.98	1707.2
	mult32	64/64	8340	53	5723.3	1.87	7868.3
EPFL* arith-metic	add128	256/129	1297	28	933.4	0.96	825.0
	bar	135/128	2688	14	1267.8	0.92	1753.1
	log2	32/32	38540	419	21480.6	14.26	40410.0
	max	512/130	2686	549	1646.3	15.81	1971.2
	mult64	128/128	33242	326	16447.3	9.47	31105.0
	sine	24/25	7044	180	4112.1	5.93	5968.1
	sqrt	128/64	21951	4591	13464.1	216.92	58397.0
	square	64/128	20030	296	12801.8	7.96	18836.0
ISCAS85	c880	60/26	313	22	198.2	0.59	129.8
	c1355	41/32	390	16	235.9	0.56	260.1
	c1908	33/25	367	25	229.6	0.86	222.6
	c2670	233/140	579	17	385.2	0.68	325.4
	c3540	50/22	937	32	521.1	1.02	404.4
	c5315	178/123	1306	28	720.3	0.72	643.7
	c7552	207/108	1469	26	903.6	1.43	898.8

* The large benchmark *hyp* is omitted and cannot be handled by both our and baseline methods. The *div* benchmark is omitted since there is no space of approximation under the given MaxED bounds.

The benchmarks used in our experiments are listed in Table I, which includes circuit names, PI/PO numbers, AIG size, AIG depth, circuit area, circuit delay, and circuit power. They are the benchmarks used in MECALS [27], EPFL arithmetic benchmarks [44], and ISCAS85 benchmarks [45]. The initial AIGs have been well optimized to ensure as little redundancy as possible. These AIGs are then used as input to our ALS flow and those ALS flows for comparison. The baseline methods are the MECALS [27] and MUSCAT [6] methods. MECALS is a state-of-the-art ALS method, in which the maximum error checking problem is converted into a SAT sweeping problem. MUSCAT is another state-of-the-art method that converts the ALS problem under maximum error constraint into a MUS problem and solves it using a MUS solver. MECALS can handle both constant [8] and SASIMI [9] LACs, while MUSCAT only supports constant LACs. In our experiments, for a circuit with N nodes, our method and MECALS consider $2N$ constant LACs and $O(N^2)$ SASIMI LACs, while MUSCAT only considers the $2N$ constant LACs.

B. Experiments Under the MaxED Constraint

This set of experiments tests the arithmetic benchmarks used in MECALS and from the EPFL benchmark suite in Table I under the MaxED constraint. Note that MaxED is a suitable error metric for arithmetic circuits, because from Eq. (2), the deviation function of MaxED considers the different significance of different POs, which measures the absolute difference between the numerical values encoded by the POs of accurate and approximate circuits. In what follows, we first compare our flow with the state-of-the-art methods on the benchmarks used in MECALS and then

TABLE II

COMPARISON OF OUR METHOD WITH THE STATE-OF-THE-ART METHODS UNDER THE MaxED CONSTRAINT. **BOLD** ENTRIES INDICATE THE SMALLEST AREA, DELAY, AND POWER RATIOS, OR THE SHORTEST RUNTIME. N/A MEANS MUSCAT CANNOT OBTAIN FINAL APPROXIMATE CIRCUITS IN 24 HOURS.

Circuit	MaxED bound	Area ratio			Delay ratio			Power ratio			Runtime/s		
		Ours	MECALS	MUSCAT	Ours	MECALS	MUSCAT	Ours	MECALS	MUSCAT	Ours	MECALS	MUSCAT
absdiff	1	**64.0%**	65.2%	86.2%	**107.2%**	108.9%	94.6%	68.1%	**65.2%**	83.7%	**0.2**	0.4	1.1
	3	**59.7%**	69.2%	84.7%	**95.1%**	106.2%	105.7%	**59.5%**	62.4%	70.3%	1.7	**0.6**	1.1
add8	1	**76.0%**	92.5%	95.0%	**77.3%**	88.1%	101.6%	**81.4%**	92.3%	94.3%	0.12	**0.11**	0.7
	3	**72.2%**	81.7%	85.5%	**76.7%**	95.7%	87.0%	**76.7%**	80.7%	85.6%	0.7	**0.2**	1.0
add32	9	**70.3%**	71.9%	84.1%	95.0%	99.6%	**76.3%**	**77.6%**	79.6%	100.0%	**28**	50	23
	97	63.0%	**62.5%**	82.6%	84.9%	86.0%	**83.7%**	69.5%	**68.5%**	82.6%	114	84	**28**
binsqrd	3	**78.4%**	79.9%	99.9%	95.2%	**93.8%**	100.0%	**83.1%**	84.0%	100.4%	**15**	1432	377
	12	**76.9%**	78.6%	97.2%	95.2%	**93.9%**	99.7%	**81.6%**	82.7%	98.2%	**271**	8947	7230
buttfly	10	**79.4%**	86.9%	97.7%	**81.6%**	91.8%	99.3%	**81.5%**	83.6%	93.1%	15	3.6	**2.2**
	111	**74.6%**	85.3%	94.9%	101.1%	**83.7%**	99.3%	**77.3%**	78.8%	86.9%	56	5.8	**2.3**
mac	1	87.1%	**86.6%**	96.6%	**94.4%**	95.8%	99.6%	92.7%	**91.3%**	92.4%	**0.3**	1.0	1.9
	2	85.1%	**82.8%**	89.4%	94.4%	91.5%	**87.4%**	88.3%	87.8%	**82.4%**	**0.8**	4.5	4.5
mult8	3	**74.1%**	75.1%	98.2%	**81.1%**	82.0%	100.0%	75.1%	**72.4%**	98.6%	**1.3**	78	949
	9	**72.5%**	73.1%	96.3%	85.9%	**81.9%**	100.0%	76.3%	**70.8%**	97.3%	**7.7**	111	7225
Average of above		**73.8%**	77.9%	92.0%	**90.4%**	92.8%	95.3%	**77.8%**	78.6%	90.4%	**37**	766	1132
mult16	9	**98.3%**	99.7%	N/A	**98.1%**	100.7%	N/A	**99.0%**	99.9%	N/A	**9.3**	154	N/A
	84	**93.7%**	95.7%	N/A	**92.1%**	97.1%	N/A	**94.5%**	96.0%	N/A	**83**	368	N/A
mult32	84	**98.0%**	99.6%	N/A	**97.5%**	102.0%	N/A	**98.2%**	99.1%	N/A	**203**	997	N/A
	7131	**93.9%**	98.6%	N/A	100.9%	**93.8%**	N/A	**94.7%**	97.8%	N/A	**1544**	3364	N/A
Average of all		**78.7%**	82.5%	N/A	**91.9%**	94.0%	N/A	**82.0%**	82.9%	N/A	**131**	867	N/A

show the scalability of our flow using the EPFL arithmetic benchmarks.

1) Comparison with State-of-the-Art Methods: We compare our ALS flow with MECALS and MUSCAT. The tested benchmarks are those used in MECALS. We run the open-source codes of MECALS and MUSCAT on the same platform for fair comparison. The MaxED bounds are set to $\lfloor 2^{O/10} \rfloor$ and $\lfloor 2^{O/5} \rfloor$ for each benchmark, where O is the number of POs of the benchmark, and the function $\lfloor x \rfloor$ gives the greatest integer less than or equal to x. In each iteration of our flow, the LAC generation step (see Fig. 2) produces both constant and SASIMI LACs.

Table II compares the area ratio, delay ratio, power ratio, and runtime of our flow, MECALS, and MUSCAT under the MaxED constraint. The bold entries indicate the smallest area ratio, delay ratio, power ratio, and runtime among the three methods, and we use the same highlighting in the following tables. We can see that our method achieves the smallest area ratios for most benchmarks. For the first 7 smaller benchmarks, our method achieves an average area, delay, and power ratio of 73.8%, 90.4%, and 77.8%, respectively. Compared to MECALS, our method reduces area, delay, and power by 4.1%, 2.4% and 0.8% on average. Compared to MUSCAT, our method reduces area, delay, and power by 18.2%, 4.9%, and 12.6% on average. Moreover, MUSCAT cannot obtain the final approximate circuit in 24 hours for the benchmarks *mult16* and *mult32*, while our method and MECALS can. Over all benchmarks, our method reduces area, delay, and power by 3.8%, 2.1%, and 0.9% on average compared to MECALS. Note that for the benchmarks *add8* and *mult16*, our method achieves smaller area, delay, and power ratios than MECALS for both MaxED bounds. However, for the benchmark *add32* under the MaxED bound of 97 and the benchmark *mac*, our method is worse than MECALS in terms of the area ratio. One possible reason is as follows. Both our flow and MECALS iteratively simplify the circuit. In each iteration, our flow selects multiple promising LACs (see Section IV-C2), while MECALS only selects one. This difference may lead to better performance of MECALS on

some benchmarks. However, our flow reduces more area than MUSCAT for all benchmarks. This is because our flow can handle more complex LACs than MUSCAT, which can achieve better approximate circuits.

Our method is more efficient than MECALS and MUSCAT. Over the first 7 benchmarks, our method is $20.7\times$ faster than MECALS and $30.6\times$ faster than MUSCAT on average. Over all benchmarks, our method speeds up by $6.6\times$ on average compared to MECALS. Note that our method is slower than MECALS for some benchmarks. For the small benchmarks *absdiff* and *add8*, the total runtime of our method is within 2 seconds, so the runtime difference is negligible. For the benchmark *buttfly*, compared to MECALS, our method takes more time but dramatically reduces the area, which is worth the trade-off. For the benchmark *add32* under the MaxED bound of 97, our method consumes more time than MECALS and MUSCAT, while the area and delay ratios are still competitive. Additionally, across all benchmarks in Table II, our method finishes after an average of 5.8 iterations. The average number of LACs applied per iteration is 7.4, obtained by dividing the total number of applied LACs across all iterations and benchmarks over the total number of iterations across all benchmarks. In contrast, MECALS needs an average of 8.3 iterations with only one LAC applied per iteration. Applying multiple LACs per iteration reduces the number of iterations, thereby shortening the overall runtime.

2) Experiments on EPFL Benchmarks: To show the scalability of our ALS flow, we test it on the large EPFL arithmetic benchmarks. The MUSCAT method cannot handle them, so we do not compare our method with MUSCAT and we only compare our method with MECALS. Similarly, the MaxED bounds are set to $\lfloor 2^{O/10} \rfloor$ and $\lfloor 2^{O/5} \rfloor$ for each benchmark, where O is the PO number of the benchmark. To accelerate our flow, we first apply the constant LACs to quickly simplify the circuit, followed by the SASIMI LACs for further simplification. Specifically, the constant LACs are first applied to the circuit POs (like truncation) until the MaxED bound is reached and then to the internal

TABLE III
COMPARISON OF OUR ALS FLOW WITH THE MECALS METHOD ON THE EPFL ARITHMETIC BENCHMARKS UNDER THE MAXED CONSTRAINT.
BOLD ENTRIES INDICATE SMALLER AREA, DELAY, AND POWER RATIOS, OR SHORTER RUNTIME. N/A MEANS THAT MECALS CANNOT OBTAIN
THE FINAL APPROXIMATE CIRCUIT IN 24 HOURS.

Circuit	MaxED bound	Area ratio		Delay ratio		Power ratio		Runtime/s	
		Ours	MECALS	Ours	MECALS	Ours	MECALS	Ours	MECALS
add128	7.6E+03	**86.9%**	94.2%	105.4%	**96.4%**	**86.6%**	92.0%	**7.2**	474
	5.8E+07	**77.6%**	84.1%	**99.9%**	100.1%	**77.1%**	81.2%	**11**	3264
bar	7.1E+03	**97.5%**	97.9%	**99.9%**	100.6%	**95.1%**	96.0%	**2.2**	412
	5.1E+07	96.2%	**95.7%**	**100.2%**	100.4%	**90.8%**	90.3%	**2.6**	753
max	8.2E+03	94.2%	94.2%	82.4%	82.4%	90.5%	**90.1%**	**273**	6362
	6.7E+07	**93.1%**	93.2%	**80.8%**	80.9%	**87.5%**	87.9%	**303**	8580
mult64	7.1E+03	**96.1%**	99.1%	100.6%	**98.5%**	**98.3%**	101.0%	**4347**	45437
	5.1E+07	**95.7%**	98.2%	102.4%	**99.2%**	**97.8%**	99.6%	**1996**	78970
square	7.1E+03	**92.9%**	99.4%	**93.3%**	98.9%	**96.3%**	99.1%	**1504**	53925
Average of above		**92.2%**	95.1%	96.1%	**95.3%**	**91.1%**	93.0%	**938**	22020
square	5.1E+07	93.0%	N/A	89.8%	N/A	95.7%	N/A	**13903**	N/A
log2	9.0E+00	94.1%	N/A	104.8%	N/A	91.9%	N/A	**37143**	N/A
	8.4E+01	93.9%	N/A	103.7%	N/A	90.9%	N/A	**36754**	N/A
sin	5.0E+00	94.9%	N/A	110.2%	N/A	94.6%	N/A	**1195**	N/A
	3.2E+01	79.7%	N/A	96.7%	N/A	72.0%	N/A	**22293**	N/A
sqrt	8.4E+01	81.2%	N/A	81.3%	N/A	77.7%	N/A	**400**	N/A
	7.1E+03	62.6%	N/A	62.9%	N/A	56.4%	N/A	**363**	N/A
Average of all		89.3%	N/A	94.6%	N/A	87.4%	N/A	**7531**	N/A

nodes of the circuit until the MaxED bound is reached. Finally, the SASIMI LACs are applied until the MaxED bound is reached. We emphasize that despite the above modification, the ALS flow is still based on the general framework presented in Fig. 2. This modification is introduced only to reduce runtime on large benchmarks, where directly considering all constant and SASIMI LACs in each iteration is computationally expensive (> 24 hours). With the modification, we reduce the number of LACs to be checked while still following the same ALS flow in Fig. 2.

Table III compares the area ratio, delay ratio, power ratio, and runtime of our flow and MECALS on the EPFL arithmetic benchmarks under the MaxED constraint. We can see that our method can handle all benchmarks in the table with an average runtime of 7531 seconds, while MECALS cannot handle the benchmarks *log2*, *sin*, *sqrt*, and *square* (under 5.1×10^7 MaxED bound) in 24 hours. For the benchmarks that both our method and MECALS can handle (the top part of the table), our method further reduces the area by 2.9% and power by 1.9% on average with a small delay overhead, while being $23.5\times$ faster than MECALS on average. Notably, for the benchmark *add128*, under the two MaxED bounds of 7.6×10^3 and 5.8×10^7, our method speeds up by $65.8\times$ and $296.7\times$, respectively, compared to MECALS and reduces area by 7.3% and 6.5%, respectively, and power by 5.4% and 4.1%, respectively. Only for the benchmark *bar* under the MaxED bound of 5.1×10^7, our method is slightly worse than MECALS in terms of the area ratio. However, our method is far more efficient than MECALS with a competitive delay ratio. Moreover, across the cases where both methods succeed (the top part of Table III), our method finishes after an average of 3.8 iterations, and the average number of LACs applied per iteration is 11.4. In contrast, MECALS needs an average of 37.6 iterations with only one LAC applied per iteration. This again shows the efficiency of our method.

C. Experiments Under the MaxHD Constraint

This set of experiments approximates the ISCAS85 and EPFL arithmetic benchmarks in Table I under the MaxHD constraint. We compare our ALS flow with MECALS. We modify the open-source codes of MECALS to support the MaxHD constraint and run both methods on the same

platform for fair comparison. MUSCAT is not compared in this experiment as its open-source code does not support the MaxHD constraint. The MaxHD bounds are set to $\lfloor O/10 \rfloor$ and $\lfloor O/5 \rfloor$ for each benchmark, where O is the number of POs of the benchmark. For the *max* benchmark, we set the parameter K in the sorting strategy in Section IV-C3 to 1000 to achieve better approximate circuits with lower hardware costs, while for the others, K still keeps the default value of 100. Similar as in Section V-B2, to accelerate our flow, the constant LACs are first applied until the MaxHD bound is reached, and then the SASIMI LACs are applied until the MaxHD bound is reached.

Table IV compares the area ratio, delay ratio, power ratio, and runtime of our flow and MECALS. Our method completes all benchmarks within 7 hours, while MECALS fails to process *sqrt* within 24 hours. For *log2*, MECALS cannot obtain the final approximate circuit due to out of memory. Under the given MaxHD bounds, our flow cannot apply any LAC to simplify *log2* and therefore terminates after one iteration, with the one-round runtime reported in Table IV. Our method achieves smaller area and delay ratios on most benchmarks, and smaller power ratios on all benchmarks. Excluding *log2* and *sqrt*, our method achieves average area, delay, and power ratios of 75.8%, 83.6%, and 74.0%, respectively, with an average runtime of 517 seconds. Compared to MECALS, our method reduces area, delay, and power by 9.4%, 5.6%, and 11.1% on average, respectively. Notably, for benchmark *c7552*, both area and power savings exceed 50%. For benchmark *c1908* under the MaxHD bound of 5 and benchmark *c2670* under the MaxHD bound of 28, although our method is worse than MECALS in terms of the area ratio, it reduces delay and power consumption. Moreover, our method is more efficient than MECALS, accelerating over MECALS by $11.7\times$ on average. The efficiency is attributed to the reduced number of iterations by our method. Across all benchmarks in Table IV, our method finishes after an average of 5.6 iterations, and the average number of LACs applied per iteration is 21.3. In contrast, MECALS needs an average of 48.9 iterations with only one LAC applied per iteration.

D. Effectiveness of Simulation and Parameter Study

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2026.3654911

10

TABLE IV
COMPARISON OF OUR ALS FLOW WITH THE MECALS METHOD ON THE ISCAS AND EPFL ARITHMETIC BENCHMARKS UNDER THE MAXHD
CONSTRAINT. **BOLD** ENTRIES INDICATE SMALLER AREA, DELAY, AND POWER RATIOS, OR SHORTER RUNTIME. N/A MEANS THAT MECALS
CANNOT OBTAIN THE FINAL APPROXIMATE CIRCUIT IN 24 HOURS. OOM MEANS THAT MECALS TERMINATES DUE TO OUT OF MEMORY.

Circuit	MaxHD bound	Area ratio		Delay ratio		Power ratio		Runtime/s	
		Ours	MECALS	Ours	MECALS	Ours	MECALS	Ours	MECALS
c880	2	**88.3%**	97.3%	**100.5%**	105.6%	**85.9%**	97.6%	**0.9**	1.9
	5	**67.5%**	92.4%	113.1%	**106.8%**	**55.6%**	95.3%	**1.1**	2.8
c1355	3	**10.1%**	10.4%	3.4%	3.4%	**8.1%**	8.3%	**0.6**	73
	6	**9.1%**	10.8%	3.4%	3.4%	**7.3%**	8.6%	**0.4**	17
c1908	2	**89.9%**	97.3%	**89.5%**	101.1%	**89.7%**	101.0%	**1.0**	5.0
	5	73.6%	**65.5%**	**84.3%**	86.6%	**62.2%**	75.3%	**1.5**	53
c2670	14	**64.2%**	84.9%	**62.9%**	99.6%	**66.2%**	83.4%	29	**26**
	28	42.3%	**41.2%**	**36.9%**	54.4%	**42.8%**	44.9%	32	80
c3540	2	**93.8%**	95.5%	**92.8%**	109.1%	**94.2%**	97.6%	**1.3**	64
	4	**92.1%**	95.6%	**98.7%**	102.8%	**91.8%**	94.6%	**1.6**	105
c5315	12	**92.7%**	96.5%	104.2%	**103.5%**	**94.0%**	96.8%	**25**	158
	24	**84.6%**	90.5%	**93.8%**	99.6%	**85.7%**	91.7%	**34**	877
c7552	10	**41.2%**	95.8%	**77.6%**	99.8%	**40.6%**	96.2%	**16**	193
	21	**32.7%**	84.2%	**68.2%**	100.1%	**29.6%**	83.9%	**29**	627
add128	12	**92.0%**	92.4%	113.2%	**96.3%**	**89.3%**	90.2%	**8.2**	9.7
	25	**83.7%**	88.9%	101.7%	**101.3%**	**80.4%**	84.8%	35	**16**
bar	12	**97.0%**	99.5%	101.0%	**98.8%**	**94.5%**	99.2%	**27**	13761
	25	**90.8%**	98.5%	99.8%	**95.8%**	**85.0%**	95.0%	**35**	18128
log2	3	100.0%	OOM	100.0%	OOM	100.0%	OOM	**22045**	OOM
	6	100.0%	OOM	100.0%	OOM	100.0%	OOM	**24133**	OOM
max	13	**86.6%**	94.2%	**74.7%**	83.1%	**82.7%**	89.9%	260	**38**
	26	**77.0%**	92.7%	**64.7%**	80.3%	**71.7%**	86.7%	172	**79**
mult64	12	**95.9%**	99.1%	98.9%	98.9%	**97.1%**	100.9%	**357**	13932
	25	**94.9%**	98.6%	**95.2%**	97.9%	**96.4%**	100.2%	**2192**	22736
sin	2	**93.6%**	97.9%	104.9%	**101.5%**	**93.7%**	97.9%	3002	**1436**
	5	**92.8%**	97.1%	**100.9%**	102.0%	**92.7%**	96.9%	4832	**2062**
sqrt	6	81.2%	N/A	82.2%	N/A	78.1%	N/A	**560**	N/A
	12	65.2%	N/A	64.8%	N/A	58.9%	N/A	**753**	N/A
square	12	**92.9%**	99.0%	**95.1%**	95.5%	**94.7%**	98.4%	**284**	37073
	25	**91.8%**	98.5%	93.4%	**92.4%**	**92.6%**	97.7%	**2075**	45644
Average w/o log2 & sqrt		**75.8%**	85.2%	**83.6%**	89.2%	**74.0%**	85.1%	**517**	6046

This section evaluates the effectiveness of simulation in our ALS flow. It also studies the impact of several important parameters in our method on the synthesis quality and runtime.

1) Effectiveness of Simulation-Guided LAC Pruning:
To show the effectiveness of the simulation-guided LAC pruning (see Section IV-B) in our ALS flow, we conduct an ablation study on the arithmetic circuits used in MECALS in Table I under the MaxED constraint. We compare our flows with and without the simulation-guided LAC pruning. For the flow without the pruning, we evaluate all candidate LACs in each iteration and select the first K valid LACs to simplify the circuit. Similar to the previous experiments, we choose MaxED bounds of $\lfloor 2^{O/10} \rfloor$ and $\lfloor 2^{O/5} \rfloor$ for each benchmark, where O is the number of POs of the benchmark. Since the flow without the pruning is very slow, we only use the simple constant LACs in this experiment to ensure that the experiment finishes in a reasonable time.

Table V compares our ALS flows with and without the simulation-guided LAC pruning under the MaxED constraint in terms of the area ratio, delay ratio, runtime, and the number of SAT problems solved in the flow. We can see that with pruning applied, the number of SAT problems solved in our flow is dramatically reduced by 98.7% on average, leading to an average runtime reduction of 96.2%. Meanwhile, applying the pruning almost does not affect the area and delay ratios of the approximate circuits. This is because the pruning just removes the invalid LACs according to the simulation results, and the valid LACs are still preserved in the design space, ensuring good qualities of the approximate circuits. An exception is the benchmark *absdiff* under the MaxED bound of 3,

where the area ratio with the pruning is much larger than that without the pruning, while the delay ratio with the pruning is much smaller than that without the pruning. We believe that this is caused by an area-delay trade-off of the technology mapping process, since the final approximate AIGs produced with and without the pruning before the technology mapping has similar size (*i.e.*, 116 with pruning versus 115 without pruning) and the same depth (*i.e.*, 12).

2) Impact of Simulation Count on Synthesis Quality and Runtime: We study the impact of the simulation count (M_{small} and M in Section IV-B1) on both synthesis quality and runtime. Recall that our method adopts a two-phase simulation-guided LAC pruning (see Section IV-B1). First, M_{small} patterns are used for rough pruning of LACs, followed by a larger set of M patterns for further filtering. We evaluate the impact of M and M_{small} on four benchmarks, *binsqrd, buttfly, mult16*, and *mult32*, under a fixed MaxED constraint of $\lfloor 2^{O/5} \rfloor$, where O is the PO number of the benchmark. For each benchmark, we test seven (M_{small}, M) pairs: $(2^5, 2^8)$, $(2^5, 2^{11})$, $(2^5, 2^{13})$, $(2^5, 2^{16})$, $(2^{10}, 2^{11})$, $(2^{10}, 2^{13})$, and $(2^{10}, 2^{16})$. Note that $(2^{10}, 2^{13})$ is the default setting in all previous experiments.

Fig. 5 shows the impact of the simulation count on the final circuit area and runtime of our method. For all benchmarks, under the same M_{small}, the final area decreases or remains unchanged as M increases. This is because a larger M can filter out more invalid LACs, thereby retaining more valid ones in the top K candidates for further evaluation, which produces smaller approximate circuits. Meanwhile, under the same M_{small}, the runtime generally increases with M, since more simulation patterns require more simulation time. An exception occurs for *mult32* with $M_{small} = 2^5$,

TABLE V
COMPARISON OF OUR ALS FLOWS WITH AND WITHOUT THE SIMULATION-GUIDED LAC PRUNING UNDER THE MaxED CONSTRAINT. ONLY
CONSTANT LACS ARE USED IN THIS EXPERIMENT. **BOLD** ENTRIES INDICATE SMALLER AREA OR DELAY RATIOS, SHORTER RUNTIME, OR FEWER
SAT PROBLEMS SOLVED.

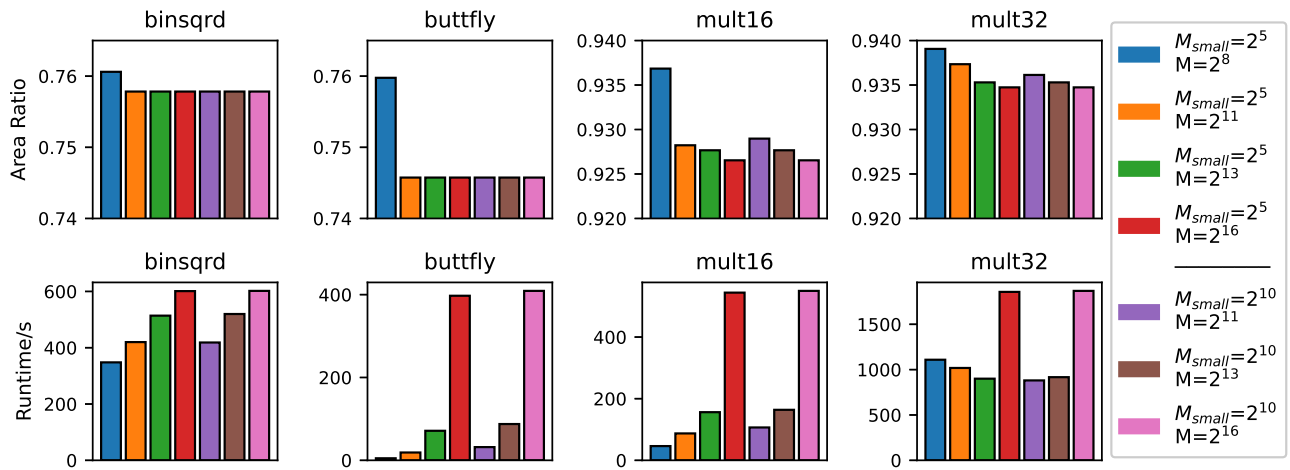| Circuit	MaxED bound	Area ratio		Delay ratio		Runtime/s		Solved #SAT	
		With pruning	W/o pruning	With pruning	W/o pruning	With pruning	W/o pruning	With pruning	W/o pruning
absdiff	1	81.7%	81.7%	88.7%	88.7%	**0.1**	1.4	**10**	518
	3	78.0%	**53.6%**	**77.9%**	94.2%	**0.1**	1.7	**10**	492
add8	1	86.1%	86.1%	91.2%	91.2%	**0.03**	0.1	**0**	122
	3	72.2%	72.2%	77.0%	77.0%	**0.05**	0.2	**4**	212
add32	9	72.0%	72.0%	98.3%	98.3%	**0.2**	5.7	**18**	952
	97	63.5%	63.5%	88.3%	88.3%	**0.4**	4.9	**39**	874
binsqrd	3	77.7%	77.7%	94.2%	94.2%	**52**	1233	**20**	6180
	12	77.0%	77.5%	94.2%	94.2%	**343**	1362	**67**	6134
buttfly	10	78.2%	78.2%	110.7%	110.7%	**0.3**	1.1	**36**	1046
	111	75.5%	75.5%	113.2%	113.2%	**0.3**	1.1	**58**	1028
mac	1	92.6%	92.6%	100.8%	100.8%	**0.1**	1.3	**4**	554
	2	85.7%	85.7%	95.5%	95.5%	**0.1**	1.2	**9**	532
mult8	3	73.9%	73.9%	85.6%	85.6%	**0.4**	71	**4**	2560
	9	72.5%	72.3%	**80.5%**	81.0%	**0.8**	106	**27**	2532
mult16	9	99.3%	99.3%	100.5%	**100.0%**	**3.5**	1208	**15**	7886
	84	**93.6%**	93.7%	**91.7%**	91.8%	**57**	1357	**124**	7700
mult32	84	98.4%	98.4%	93.5%	95.2%	**162**	14549	**111**	15872
	7131	**94.5%**	94.7%	**94.8%**	101.2%	**669**	14437	**417**	15398
Average		81.8%	**80.5%**	**93.2%**	94.5%	**72**	1909	**54**	3922

Fig. 5. Impact of simulation count on final area after applying our method and runtime of our method under $\lfloor 2^{O/5} \rfloor$ MaxED constraint. Runtime data differs from that in Table II since a different computer is used in this experiment.

where runtime first decreases and then increases. When both M_{small} and M are small, pruning is ineffective, and many invalid LACs remain in the top K LACs and must be verified by SAT solving, which dominates the runtime. As M increases from 2^8 to 2^{13}, more invalid LACs are eliminated by simulation, leading to fewer SAT problems and reduced runtime. When M further increases to 2^{16}, the runtime rises again due to the long simulation time. Overall, the default setting of $(M_{small}, M) = (2^{10}, 2^{13})$, shown as brown bars in Fig. 5, provides a good trade-off between the final area and runtime across the four benchmarks.

3) Impact of Simulation Count on SAT Solving Statistics:
We further study how the simulation count influences the SAT solving statistics. In this study, we fix $M_{small} = 2^5$ and vary $M \in \{2^8, 2^{11}, 2^{13}, 2^{16}\}$, while keeping the other settings identical to those in the previous experiment in Section V-D2. Table VI shows, for different M values, the total number of SAT problems solved, the percentages of SAT and UNSAT results, and the number of UNDEFINED results observed in the first two iterations of our method. Since different M values lead to different iteration numbers, we restrict the comparison to the first two iterations for fairness. As M increases, the percentage of SAT results generally decreases while that of UNSAT results generally

increases. This trend is expected because a larger M prunes more invalid LACs, which increases the likelihood that the preserved top K LACs are valid. As valid LACs correspond to UNSAT results when their maximum errors are formally checked, the UNSAT ratio rises, while the SAT ratio decreases. Moreover, UNDEFINED results are only observed in *mult32* when $M = 2^{11}, 2^{13}$, and 2^{16}. Due to the large size of *mult32*, the SAT solver may fail to resolve some instances within the computation budget.

4) Impact of Number of Top LACs (Parameter K):
Recall that during the LAC selection step shown in Fig. 4, only the top K LACs are kept for further evaluation based on their maximum error lower bounds and estimated area reductions. We study the impact of K on the final area, runtime, and iteration number of our method. Specifically, we test $K \in \{50, 100, 1000, 10000\}$ on the four benchmarks *binsqrd, buttfly, mult16,* and *mult32* under the MaxED constraint of $\lfloor 2^{O/5} \rfloor$. The results are shown in Table VII.

The impact of K on the final area and runtime varies across benchmarks, but a common related trend can be observed. As K increases, the number of iterations decreases, since more candidate LACs are evaluated and applied in each iteration. However, a larger K with fewer iterations does not necessarily yield shorter runtime, because evalu-

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2026.3654911

12

TABLE VI
IMPACT OF SIMULATION COUNT ON SAT SOLVING STATISTICS IN THE FIRST TWO ITERATIONS OF OUR METHOD UNDER $\lfloor 2^{O/5} \rfloor$ MAXED CONSTRAINT. M_{small} IS FIXED TO 2^5.

Circuit	M	All SAT calls	SAT percentage	UNSAT percentage	#UNDEF count
binsqrd	2^8	98	42.9%	57.1%	0
	2^{11}	60	3.3%	96.7%	0
	2^{13}	60	3.3%	96.7%	0
	2^{16}	60	3.3%	96.7%	0
buttfly	2^8	100	95.0%	5.0%	0
	2^{11}	100	95.0%	5.0%	0
	2^{13}	100	95.0%	5.0%	0
	2^{16}	36	80.6%	19.4%	0
mult16	2^8	17	17.6%	82.4%	0
	2^{11}	15	0.0%	100.0%	0
	2^{13}	15	0.0%	100.0%	0
	2^{16}	15	0.0%	100.0%	0
mult32	2^8	106	42.5%	57.5%	0
	2^{11}	69	5.8%	88.4%	4
	2^{13}	33	6.1%	90.9%	1
	2^{16}	33	6.1%	90.9%	1

TABLE VII
IMPACT OF NUMBER OF TOP LACS SELECTED (K) ON FINAL AREA, RUNTIME, AND NUMBER OF ITERATIONS OF OUR METHOD UNDER $\lfloor 2^{O/5} \rfloor$ MAXED CONSTRAINT. RUNTIME DATA DIFFERS FROM THAT IN TABLE II SINCE A DIFFERENT COMPUTER IS USED IN THIS EXPERIMENT.

Circuit	K	Final Area/μm^2	Runtime/s	#Iterations
binsqrd	50	801.7	562	10
	100	797.5	445	7
	1000	795.6	851	5
	10000	790.3	927	5
buttfly	50	127.2	48	7
	100	127.2	73	7
	1000	127.2	49	5
	10000	127.2	32	4
mult16	50	1327.3	149	12
	100	1316.2	142	7
	1000	1323.1	123	5
	10000	1328.9	729	4
mult32	50	5379.1	1240	19
	100	5353.0	866	10
	1000	5360.4	1377	7
	10000	5382.2	2924	4

ating more LACs per iteration also requires solving more SAT problems per iteration. Thus, the overall runtime is determined by the trade-off between the iteration count and the number of SAT calls per iteration. For *binsqrd*, *mult16*, and *mult32*, runtime decreases at first as K increases, but then rises again when K further increases. For *buttfly*, runtime exhibits the opposite trend, first increasing and then decreasing.

The effect of K on the final circuit area also differs by benchmarks. For *binsqrd*, the final area consistently decreases with K. For *buttfly*, the final area remains the same for different K values. For *mult16* and *mult32*, the final area first decreases and then increases as K grows. One possible reason is that a small K may exclude promising LACs from further evaluation, leading to suboptimal results with larger area. Conversely, when K is very large, too many LACs are applied in each iteration under the greedy selection strategy (see Fig. 4). This causes the circuit to deviate significantly from the current approximate circuit, making the previously computed maximum error lower bounds of LACs inaccurate. Guided by these inaccurate

values, some poor LACs with large errors may be applied, which eventually causes the flow to terminate prematurely and produces a circuit with larger area.

E. Comparison on Approximate Adder and Multiplier Designs

Given the importance of approximate adders and multipliers, especially in AI accelerators, we compare the approximate adders and multipliers synthesized by our ALS flow with those from the EvoApproxLib [46] (version 2022), a widely-used open-source library of approximate adders and multipliers. The benchmarks compared are the 12-bit and 16-bit unsigned adders and the 8-bit, 11-bit, 12-bit, and 16-bit unsigned multipliers. The approximate designs from the EvoApproxLib are those synthesized under the MaxED constraint. Our ALS flow starts from the accurate circuits from the EvoApproxLib, converts them to AIGs, and applies the constant and SASIMI LACs to simplify the circuits under the MaxED constraint. The MaxED bounds are set as the MaxEDs of the approximate circuits from the EvoApproxLib, which means that the approximate circuits generated by our flow have the same or smaller MaxEDs compared to those in the EvoApproxLib.

Fig. 6 shows the comparison results, where each subfigure corresponds to an approximate adder or multiplier and plots the area ratio-MaxED and the delay ratio-MaxED curves of the approximate circuits synthesized by our flow and those from the EvoApproxLib. Comparing the results from our flow (shown in red) with those from the EvoApproxLib (shown in blue), we can see that under the same MaxED bound, the approximate circuits synthesized by our ALS flow have much smaller area and delay ratios than those from the EvoApproxLib for all benchmarks. Notably, when the MaxED is small, there is a large gap between both the area and delay ratios of the approximate circuits synthesized by our flow and those from the EvoApproxLib. This shows the effectiveness and practicality of our flow, since reducing the hardware cost and delay under a small error bound is more challenging but important for real-world applications. As the MaxED bound increases, the area and delay ratios of the approximate circuits from both our flow and the EvoApproxLib decrease due to more approximation opportunities given by the large error bounds. When the MaxED is large, the improvement of our flow over the EvoApproxLib is reduced, since the approximation opportunities are more abundant and the ALS method used for producing the EvoApproxLib can also generate good approximate circuits in this case.

VI. CONCLUSION

This paper studies ALS under the maximum error constraint. We propose to utilize logic simulation to guide the pruning of invalid LACs that violate the error constraint and the selection of promising LACs for better circuit simplification. By leveraging the simulation-guided techniques, we further propose an efficient ALS flow that iteratively applies a set of promising LACs to approximate the input circuit. The proposed flow can handle complex LACs and scale to large circuits with tens of thousands of gates. The experimental results show that our ALS flow can achieve a better trade-off between error and hardware cost compared to the state-of-the-art ALS methods.
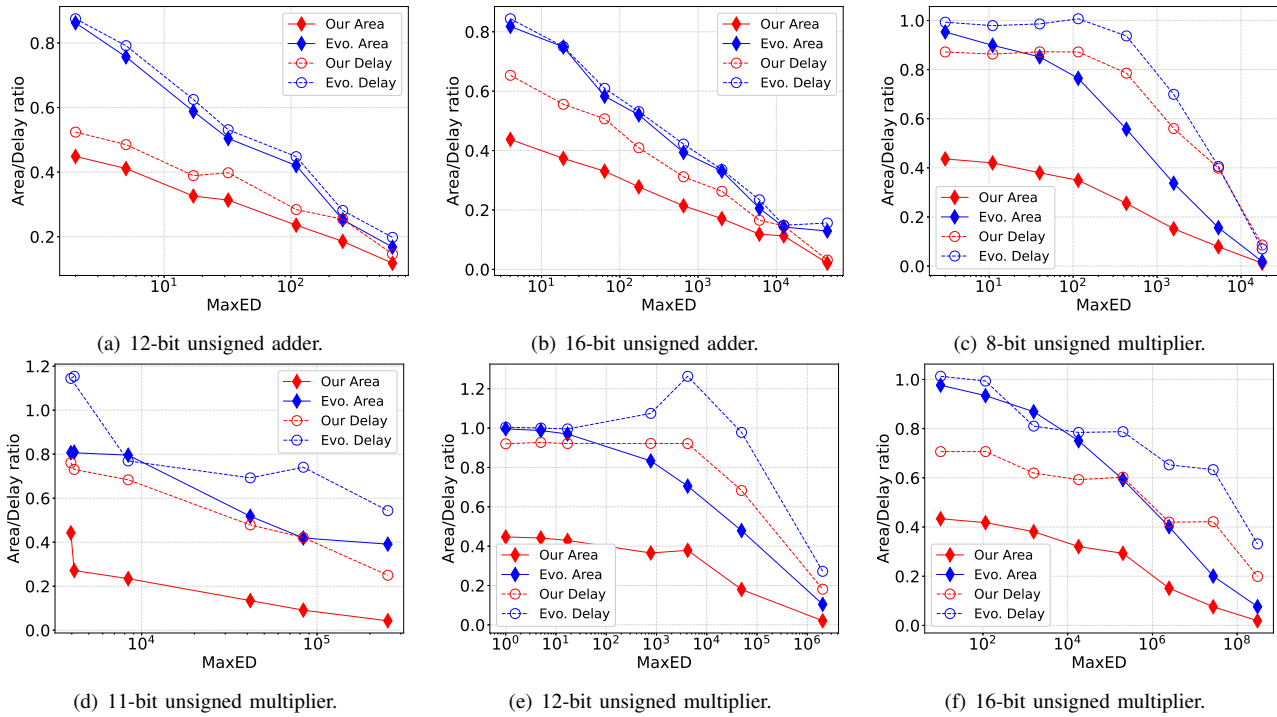
This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2026.3654911

13

Fig. 6. Comparison between approximate adders and multipliers synthesized by our flow and those from the EvoApproxLib under the MaxED constraint.

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] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–33, 2016.

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

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

[5] 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*, 2018, pp. 545–550.

[6] L. Witschen, T. Wiersema, M. Artmann, *et al.*, "MUSCAT: MUS-based circuit approximation technique," in *Design, Automation & Test in Europe Conference & Exhibition*, 2022, pp. 1–6.

[7] M. Rezaalipour, M. Biasion, I. Scarabottolo, *et al.*, "A parametrizable template for approximate logic synthesis," in *International Conference on Dependable Systems and Networks Workshops*, 2023, pp. 175–178.

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

[9] 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 Conference & Exhibition*, 2013, pp. 1367–1372.

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

[11] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, *et al.*, "Partition and propagate: An error derivation algorithm for the design of approximate circuits," in *Design Automation Conference*, 2019, pp. 1–6.

[12] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, *et al.*, "A formal framework for maximum error estimation in approximate logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[13] R. Venkatesan, A. Agarwal, K. Roy, *et al.*, "MACACO: Modeling and analysis of circuits for approximate computing," in *International Conference on Computer-Aided Design*, 2011, pp. 667–673.

[14] A. Chandrasekharan, M. Soeken, D. Große, *et al.*, "Approximation-aware rewriting of AIGs for error tolerant applications," in *International Conference on Computer-Aided Design*, 2016, pp. 1–8.

[15] M. Češka, J. Matyáš, V. Mrazek, *et al.*, "Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished," in *International Conference on Computer-Aided Design*, 2017, pp. 416–423.

[16] S.-Y. Lee, H. Riener, A. Mishchenko, *et al.*, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2021.

[17] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Design, Automation Conference*, 2016, pp. 1–6.

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

[19] Z. Zhou, Y. Yao, S. Huang, *et al.*, "DALS: Delay-driven approximate logic synthesis," in *International Conference on Computer-Aided Design*, 2018, pp. 1–7.

[20] C. Meng, Z. Zhou, Y. Yao, *et al.*, "HEDALS: Highly efficient delay-driven approximate logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3491–3504, 2023.

[21] C. Meng, W. Qian, and A. Mishchenko, "ALSRAC: Approximate logic synthesis by resubstitution with approxi-

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully

content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2026.3654911

14

mate care set," in *Design, Automation Conference*, 2020, pp. 1–6.

[22] X. Wang, Z. Yan, C. Meng, *et al.*, "DASALS: Differentiable architecture search-driven approximate logic synthesis," in *International Conference on Computer Aided Design*, 2023, pp. 1–9.

[23] 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*, vol. 41, no. 11, pp. 5085–5099, 2022.

[24] C. Meng, X. Wang, J. Sun, *et al.*, "SEALS: Sensitivity-driven efficient approximate logic synthesis," in *Design Automation Conference*, 2022, pp. 439–444.

[25] X. Wang, S. Tao, J. Zhu, *et al.*, "AccALS: Accelerating approximate logic synthesis by selection of multiple local approximate changes," in *Design Automation Conference*, 2023, pp. 1–6.

[26] J. Echavarria, S. Wildermann, O. Keszöcze, *et al.*, "Probabilistic error propagation through approximated boolean networks," in *Design Automation Conference*, 2020, pp. 1–6.

[27] C. Meng, J. Sun, Y. Mai, *et al.*, "MECALS: A maximum error checking technique for approximate logic synthesis," in *Design, Automation & Test in Europe Conference & Exhibition*, 2023, pp. 1–6.

[28] 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*, vol. 39, no. 7, pp. 1470–1483, 2019.

[29] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Design Automation Conference*, 1993, pp. 213–218.

[30] Berkeley Logic Synthesis and Verification Group, *ABC: A system for sequential synthesis and verification*, http://people.eecs.berkeley.edu/~alanmi/abc/, 2025.

[31] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *International Conference on Theory and Applications of Satisfiability Testing*, 2009, pp. 244–257.

[32] H. Riener, E. Testa, W. Haaswijk, *et al.*, "Scalable generic logic synthesis: One approach to rule them all," in *Design Automation Conference*, 2019, pp. 1–6.

[33] A. T. Calvino, H. Riener, S. Rai, *et al.*, "A versatile mapping approach for technology mapping and graph optimization," in *Asia and South Pacific Design Automation Conference*, 2022, pp. 410–416.

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

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

[36] Y. Sun, T. Liu, M. D. Wong, *et al.*, "Massively parallel AIG resubstitution," in *Design Automation Conference*, 2024, pp. 1–6.

[37] A. Costamagna, A. T. Calvino, A. Mishchenko, *et al.*, "Area-oriented optimization after standard-cell mapping," in *Asia and South Pacific Design Automation Conference*, 2025, pp. 1112–1119.

[38] A. T. Calvino, G. De Micheli, A. Mishchenko, *et al.*, "Enhancing delay-driven LUT mapping with Boolean decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[39] E. Testa, L. Amaru, M. Soeken, *et al.*, "Extending Boolean methods for scalable logic synthesis," *IEEE Access*, vol. 8, pp. 226 828–226 844, 2020.

[40] A. Mishchenko, *Introduction to ABC: An open-source synthesis and verification CAD tool for FPGAs and ASICs*, https://ethz.ch/content/dam/ethz/special-interest/itet/efcl-dam/documents/Introduction%20to%20ABC.pdf, 2019.

[41] C. Wolf, *Yosys Open SYnthesis Suite*, https://yosyshq.net/yosys/, 2025.

[42] Synopsys, Inc., *Synopsys softwares*, http://www.synopsys.com, 2025.

[43] 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*, vol. 41, no. 1, pp. 15–28, 2021.

[44] EPFL Integrated Systems Laboratory, *The EPFL combinational benchmark suite*, https://lsi.epfl.ch/page-102566-en-html/benchmarks/, 2025.

[45] M. C. Hansen, H. Yalcin, and J. P. Hayes, "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.

[46] V. Mrazek, R. Hrbacek, Z. Vasicek, *et al.*, *EvoApproxLib: Library of approximate arithmetic circuits*, https://github.com/ehw-fit/evoapproxlib/releases/tag/v1.2022/, 2022.

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).

Weikang Qian is an associate professor in the Global College 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 got the best student paper award at the International Workshop on Logic and Synthesis (IWLS) and the best paper nominations at the International Conference on Computer-Aided Design (ICCAD) and the Design, Automation, and Test in Europe Conference (DATE). 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 and Scientific Director of the EcoCloud center 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, of the Swiss Academy of Engineering Sciences, and 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 2025 IEEE Gustav Kirchhoff Award, the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards.