ALFANS: Multi-level Approximate Logic Synthesis Framework by Approximate Node Simplification

Yi Wu and Weikang Qian, *Member, IEEE*

***Abstract*— Approximate computing is an emerging design paradigm targeting at error-tolerant applications. It trades off accuracy for improvement in hardware cost and energy efficiency. In this paper, we propose ALFANS, a novel multi-level approximate logic synthesis framework by approximate node simplification. ALFANS works on the Boolean network representation of circuits. Its basic operation is to perform approximate simplification to nodes in a Boolean network. Based on this framework, we propose three different algorithms for three different types of error constraints. The first algorithm, ALFANS-ER, handles error rate constraint only. The second one, ALFANS-ER-MEM, handles a combination of error rate and maximum error magnitude constraint. The third one, ALFANS-ER-AEM, handles a combination of error rate and average error magnitude constraint. All these three algorithms repeatedly pick the single most effective node to simplify in each iteration. When only the error rate is constrained, we also propose an accelerated version, ALFANS-ER-Fast, which formulates a knapsack problem to pick multiple nodes for simplification simultaneously in each iteration. It significantly improves the runtime over ALFANS-ER with almost the same circuit area. Compared to the respective state-of-the-art approaches handling the same type of error constraint, ALFANS-ER-Fast and ALFANS-ER-MEM reduce circuit area by 1.3% and 19.5%, respectively. A salient feature of ALFANS-ER-Fast is its runtime efficiency: it has a speedup of $5.9\times$ over the state-of-the-art method.**

***Index Terms*—approximate computing, inexact circuit, approximate logic synthesis.**

I. Introduction

Approximate computing emerges as a novel computing paradigm due to two recent trends. One is the continuous pursuing of low-cost, high-performance, and low-power computing systems. The other is the prevalence of applications such as multimedia, machine learning, and data mining, which are inherently error tolerant. The key idea of approximate computing is to exploit the error tolerance of applications to introduce a small change to the function of a computing system. Although such a change incurs error, if it is proper, the latency and power consumption of the computing system can be significantly reduced [1]–[3].

The research in approximate computing crosses many layers of modern computing systems. At circuit level, it aims at deriving a circuit that not exactly but *approximately* implements the target function. Some works in this area focused on the manual design of approximate arithmetic circuits, such as adders [4]–[9] and multipliers [10]–[13]. Others focused on *approximate logic synthesis (ALS)*, which automatically synthesizes an approximate circuit for a specified Boolean function under the given error constraints [14]–[25].

To quantify the error of an approximate circuit, several error metrics have been proposed [1]. Among them, two most widely used metrics are *error rate (ER)* and *error magnitude (EM)*. ER is the probability that the outputs are wrong given an input distribution. EM depicts the error of the numerical value encoded by the outputs. It can be further classified into two types. The first is *maximum error magnitude (MEM)*, also known as the *worst-case error*. It is the maximal numerical error encoded by the outputs. The second is *average error magnitude (AEM)*, also known as the *mean error distance (MED)*. It is the mean value of all the numerical deviations of the incorrect outputs from the correct ones.

In this work, we propose ALFANS, a novel multi-level approximate logic synthesis framework by approximate node simplification. Traditional multi-level logic synthesis usually includes two phases: a technology-independent synthesis phase and a technology mapping phase [26]. ALFANS works at the first phase by manipulating the *Boolean network* representation of a circuit. Boolean network is a classic representation of logic circuits, in which each node represents a Boolean expression [26]. Simplifying the expressions of the nodes in a Boolean network helps reduce the total area of the circuit. The traditional logic synthesis methods optimize circuits while always keeping their functions unchanged. In contrast, ALFANS works more aggressively. It performs approximate simplification to the nodes in a Boolean network.

Based on the ALFANS framework, we propose three different algorithms for three different types of error constraints. The first algorithm, ALFANS-ER, handles ER constraint only. The second one, ALFANS-ER-MEM, handles a combination of ER and MEM constraint. The third one, ALFANS-ER-AEM, handles a combination of ER and AEM constraint. All these three algorithms repeatedly pick the single most effective node to simplify in each iteration. When only ER is constrained, we also propose an accelerated version, ALFANS-ER-Fast. It significantly improves the runtime over ALFANS-ER with almost the same circuit area.

In summary, our main contributions are as follows:

- We propose ALFANS, a novel approximate logic synthesis framework working at the Boolean network representation of circuits. It simplifies the factored-form expressions of the nodes in a Boolean network. Since there exist multiple ways to simplify a node, this provides us with much freedom in exploring a good solution.
- When only ER is constrained, we propose an iterative algorithm ALFANS-ER that selects a single node for simplification in each iteration. We further speed up the algorithm by proposing to choose multiple nodes for simplification simultaneously, which significantly reduces the runtime. The accelerated algorithm, ALFANS-ER-Fast, formulates a multi-state knapsack problem to determine the optimal choice. The experimental results showed that it has almost the same circuit area as ALFANS-ER, but has a $3.4\times$ runtime speedup over ALFANS-ER. Compared to a state-of-the-art method [17], ALFANS-ER-Fast could reduce the circuit area by 1.3% with a $5.9\times$ runtime speedup.
- We propose algorithm ALFANS-ER-MEM that can han-

Yi Wu and Weikang Qian was/is with the University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China, 200240.
E-mail: wuyiee@126.com, qianwk@sjtu.edu.cn

dle the combined error constraints on ER and MEM. Compared to a state-of-the-art work [19] for the same type of combined constraints, ALFANS-ER-MEM could reduce the circuit area by 19.5% on average.
- We propose algorithm ALFANS-ER-AEM that can handle the combined error constraints on ER and AEM. To the best of our knowledge, this is the first time an ALS algorithm is proposed to handle this kind of combined constraints.

A preliminary version of this work was presented in [21]. In that work, we only proposed ALS algorithms for the single constraint on ER. In this work, we further propose two ALS algorithms for the combined constraints on ER and EM.

The rest of the paper is organized as follows. Section II reviews the prior related works. Section III describes the basic idea of ALFANS and the key operations used in the proposed algorithms. Section IV presents algorithms ALFANS-ER and ALFANS-ER-Fast for the single constraint on ER. Section V presents algorithm ALFANS-ER-MEM for the combined constraints on ER and MEM. Section VI presents algorithm ALFANS-ER-AEM for the combined constraints on ER and AEM. Section VII shows the experimental results. Finally, Section VIII concludes the paper.

II. Related Works

There are two types of circuit designs, two-level and multi-level. A two-level circuit is a logic implementation based on sum-of-products (SOP) form, where the first level is composed of AND gates and the second level is composed of OR gates. Its opposite is a multi-level circuit. It could contain an arbitrary number of levels of gates and there are no restrictions on the gate types at each level. A few two-level ALS methods have been proposed recently. For example, Shin and Gupta proposed a two-level ALS method under the ER constraint only [14]. Miao *et al.* proposed another method under both the ER and MEM constraints [18]. In contrast, our work focuses on ALS for multi-level circuits. The previous two-level ALS methods cannot be directly applied to multi-level designs. This is due to the different representations of the two designs: a two-level design is usually represented as an SOP expression, while a multi-level design is typically represented as a graph such as a Boolean network [27] and an AND-inverter graph (AIG) [28].

There are also some previous works on multi-level ALS [15]–[17], [19]–[25], [29]. Some works only perform trivial approximate simplifications. For example, Shin and Gupta proposed to deliberately inject stuck-at-0 and stuck-at-1 faults into the circuit [15]. It can be viewed as a special case of the proposed approximate node simplification. Other previous works use more sophisticated techniques to perform approximation. However, most of them can only handle a single constraint. For example, Venkataramani *et al.* proposed to encode the error constraint as a function and exploit *external don't cares* to simplify the circuit [16]. However, the method is only applicable to constraint on maximum absolute or relative error magnitude. Another approach, SASIMI, identifies signal pairs of similar functionality and substitutes one with the other. However, it only works under the single constraint of either ER or AEM [17]. Liu and Zhang proposed a randomized ALS algorithm that accepts local transformations probabilistically [25]. Besides local transformations that reduce area, it also has local transformations that increase the area. Like SASIMI, it can only handle a single constraint of either ER or AEM. Froehlich *et al.* proposed an ALS algorithm that employs symbolic computer algebra for error-metric evaluation [29]. It iteratively chooses one gate and replaces the gate by one input of that gate. However, it can only handle a single constraint related to EM, such as MEM and mean square error. In contrast, we propose two algorithms that can handle the combined constraints on ER and EM. Furthermore, our basic ALFANS framework is based on simplifying the Boolean expressions of the nodes in a Boolean network. This takes a different perspective than the previous methods.

There are also some previous works that can handle combined error constraints. Chandrasekharan *et al.* proposed to perform approximate rewriting on the AIG representation of a circuit [22]. It can handle combined error constraints such as combination of ER, MEM, and maximum bit-flip error. However, it lacks a sophisticated way to select the node to apply approximate change. Its focuses on the cuts on the critical path of the circuit and chooses the one with the smallest size to approximate. Miao *et al.* proposed a multi-level ALS algorithm for the combined constraints on ER and MEM [19]. It first solves an ER-unconstrained ALS problem and then iteratively refines the resultant circuit until the ER is within the bound. Different from [19], our proposed algorithm that handles the same type of combined constraints considers the impacts of ER and MEM jointly. By doing so, our experimental results demonstrated that our algorithm generates circuits with smaller area than [19] and runs faster. Furthermore, we also propose an ALS algorithm that handles the combined constraints on ER and AEM.

III. Basic Idea and Key Operations

In this section, we introduce the basic idea of ALFANS and the key operations used in the proposed algorithms.

A. Approximate Simplified Expression

ALFANS works on the Boolean network representation of a given circuit. In traditional logic synthesis, one useful operation for optimizing Boolean network is *simplification* [27]. Each node in a Boolean network is represented both in a sum-of-product (SOP) form and a factored form. The simplification operation calls the two-level logic minimizer Espresso [30] to obtain the optimal SOP form of a node. However, the obtained SOP is accepted only if the literal count of the corresponding factored form is reduced over the original factored form. This is because for multi-level Boolean networks, the factored form provides a better area estimate for the node [27].

Inspired by the traditional simplification operation, we propose an *approximate simplification*. It simply removes some literals from the original factored expression of a node. The result is an approximation to the original expression. We call it *approximate simplified expression (ASE)*.

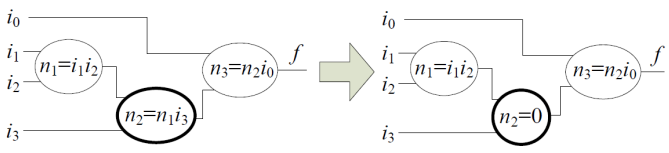


Fig. 1. A Boolean network with the original expression of n_2 replaced by the ASE $n_2 = 0$.

If we use an ASE to replace the original factored expression of a node n in a network, the Boolean function of n will change. However, among all primary input (PI) patterns that cause an error at node n, only a subset can actually cause an error at the primary outputs (POs) and the remaining cannot propagate the error at the node n to any PO. For example,

consider a Boolean network with four PIs i_0, i_1, i_2, i_3 and one PO f as shown in Fig. 1. The original factored expression of node n_2 is $n_2 = n_1 i_3$. If we use a constant 0 to replace its original expression, then two PI patterns 0111 and 1111 will cause an error at node n_2. However, the error at n_2 could be propagated to the PO f only when i_0 is 1. Thus, only the PI pattern 1111 will cause an error at PO. The other PI pattern 0111 will block the propagation of the error at n_2. To distinguish different PI patterns, we make the following definition.

Definition 1 *If we use an ASE to replace the original expression of a node n, errors will occur at the output of n. We call a PI pattern that causes an error at node n an* **Apparent Erroneous PI Pattern (AEPIP)** *of the ASE and a PI pattern that causes an error at the POs a* **Real Erroneous PI Pattern (REPIP)** *of the ASE. We call the error rate at node n the* **Apparent Error Rate (AER)** *of the ASE and the error rate of the entire network the* **Real Error Rate (RER)** *of the ASE.*

For the Boolean network shown in Fig. 1, the PI patterns 0111 and 1111 are AEPIPs of the ASE $n_2 = 0$ for node n_2, while the PI pattern 1111 is the only REPIP of the ASE.

The AER is the sum of the probabilities of all the AEPIPs, while the RER is the sum of the probabilities of all the REPIPs. It can be easily seen that an REPIP must be an AEPIP, but not the vice versa. As a result, the AER of an ASE is larger than or equal to the RER of the ASE.

It should be noted that AEPIPs, REPIPs, AERs, and RERs are defined over the current network where the ASE would be applied. The current network is not necessarily the input network. As will be shown later, our proposed algorithms are iterative and they generate a sequence of intermediate approximate networks. Thus, the current network could be any approximate network in the sequence.

On the one hand, the RER of an ASE accurately measures the contribution of an ASE to the ER increase of the entire network. It will be used in algorithms ALFANS-ER, ALFANS-ER-MEM, and ALFANS-ER-AEM. On the other hand, the AER will play an important role in algorithm ALFANS-ER-Fast. Next, we will discuss how we generate ASEs for a node in Section III-B. Then, we will show how we obtain the AER and RER of an ASE in Sections III-C and III-D, respectively. Finally, we will show how to evaluate the impacts of an ASE on MEM and AEM of an approximate circuit in Sections III-E and III-F, respectively.

B. Generating ASEs for a Node

An ASE for a node is obtained by deleting some literals from the original factored expression of the node. Assume a factored expression has N literals. Then, for any $1 \leq M \leq N$, there are in total $\binom{N}{M}$ different ways to remove M literals from the expression. If we want to remove all N literals, there are two different simplifications: one is to make the node a constant 0 and the other is to make it a constant 1.

As an example, consider a node n with its factored expression as $n = (a + b)(c + d)$, where a, b, c, and d are the immediate inputs of n. If we want to remove one literal from the expression, we have four choices: removing literal a, we get $n = b(c + d)$; removing literal b, we get $n = a(c + d)$; removing literal c, we get $n = (a + b)d$; removing literal d, we get $n = (a + b)c$. If the goal is to remove four literals, then two choices are available: $n = 0$ and $n = 1$.

The two simplifications $n = 0$ and $n = 1$ are special, because they completely delete the node n from the network. Furthermore, after logic implication, some nodes in the transitive fan-in or fan-out cone of n may also be simplified or deleted. Actually, this is same as the way that [15] applies to do simplification.

C. Computing Apparent Error Rate of an ASE

To compute the AER of an ASE for a node, we first compare it with the original expression to see which local input patterns of this node generate an erroneous output at the node. It can be easily achieved by first XORing the ASE and the original expression and then obtaining the on-set of the XORed expression. These input patterns are called the *erroneous local input patterns* as defined below.

Definition 2 *If we use an ASE to replace the original expression of a node n, errors will occur at the output of n. We call a local input pattern of n that produces a wrong value at n an* **Erroneous Local Input Pattern (ELIP)** *of the ASE.*

The AER of an ASE can be obtained by adding up the probabilities of all ELIPs of the ASE. For example, suppose that the ELIPs of an ASE are 1001, 1010, and 1011. Assume that their probabilities are 0.03, 0.01, and 0.02, respectively. Then, the AER of the ASE is 0.06. The probability of a given local input pattern can be obtained by running logic simulation on the entire circuit and counting the frequency that the specific pattern occurs. In our implementation, we do not apply all possible input combinations as stimuli in a logic simulation since this takes long time or is even infeasible for large circuits. Instead, we just apply 10000 randomly generated input patterns in a logic simulation. Note that to obtain the probabilities of all the local input patterns of interest for all the nodes, only one run of logic simulation is required.

D. Estimating Real Error Rate of an ASE

To accurately calculate the RER of an ASE, we need to replace the original expression by the ASE and then obtain the error probability of the new Boolean network. In order to obtain the RERs of all the ASEs for all the nodes in the Boolean network, the number of new networks we need to analyze equals the total number of ASEs over all the nodes in the network. This approach is very time-consuming. Instead, we propose a fast method to estimate the RER of an ASE. The estimate actually gives an *upper bound* for the RER. To estimate the RERs of all ASEs for all nodes in the network, it only requires a single analysis of the original Boolean network.

Our estimation method is based on the ELIPs of an ASE. For an internal node n in a Boolean network, the ELIPs of an ASE account for the error at the output of n. However, even in a well-optimized network, n may have satisfiability don't cares (SDCs) and observability don't cares (ODCs) due to its surrounding nodes [31]. On the one hand, if an ELIP is an SDC of n, the ELIP cannot occur at all. On the other hand, if an ELIP is an ODC of n, the wrong value at n caused by the ELIP cannot be propagated to the POs.

Therefore, to compute the RER of an ASE for a node n, we should ignore those ELIPs of the ASE that are either SDCs or ODCs of the node n. The remaining ELIPs could introduce errors to the POs when they occur. We call them *non-don't-care ELIPs* of the ASE. Our method to estimate the RER of an ASE is to add up the probabilities of all the non-don't-care ELIPs. To obtain SDCs and ODCs of a node, we use the command "mfs" in MVSIS [32]. To speed up this process, we set the window size to 2×2 and choose a SAT-based

computation method. Since we only get a subset of the SDCs and ODCs, the obtained estimate gives an upper bound on the RER of an ASE. Despite this, it is much faster than getting the exact value. Furthermore, since the estimate of RER is an upper bound, once an ASE is selected based on this estimate, the ASE is guaranteed to not violate the ER constraint. Thus, we use this method to estimate the RER.

E. Checking Maximum Error Magnitude against Threshold

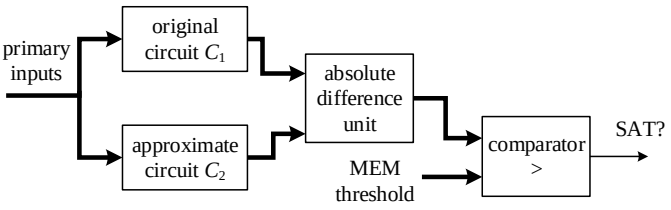


Fig. 2. The circuit for checking the maximum error magnitude of an approximate circuit against the threshold.

When the error constraint includes MEM, we should ensure that after each change due to ASE, the MEM of the approximate circuit does not exceed the given MEM threshold, denoted as D_{mem}. We apply a method similar to that used in [33]–[35] to do the check. The idea is to build a circuit shown in Fig. 2, which consists of the original circuit, an approximate circuit, an absolute difference unit, and a comparator. The original and approximate circuits are given the same primary inputs. The absolute difference unit computes the absolute difference between their output binary numbers. The comparator compares the absolute difference with the given MEM threshold. It outputs a 1 if and only if the absolute difference is larger than the threshold. We can apply a SAT solver to check whether the output of the comparator is unSAT. If so, the MEM of the approximate circuit does not exceed the threshold.

F. Calculating Average Error Magnitude

When the error constraint includes AEM, we should ensure that after each change due to ASE, the AEM of the approximate circuit does not exceed the given AEM threshold, denoted as D_{aem}. In this section, we propose a novel method for calculating the AEM of an approximate circuit.

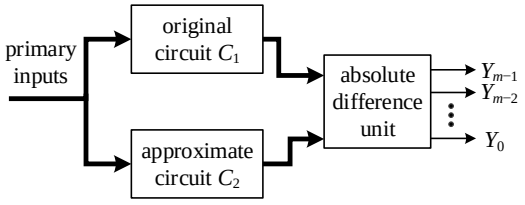


Fig. 3. The circuit for calculating the average error magnitude of an approximate circuit.

Our method first builds a circuit shown in Fig. 3. It is similar to the one shown in Fig. 2 except that the comparator is not included. Suppose the number of the primary outputs of the original circuit is m. Then, the absolute difference unit also has m outputs. Denote its m outputs as $Y_{m-1}, Y_{m-2}, \cdots, Y_0$, where Y_i $(0 \leq i \leq m-1)$ has the weight of 2^i. Then, the absolute error of the approximate circuit is $Y = \sum_{i=0}^{m-1} 2^i Y_i$. Under a random input distribution, Y_i $(0 \leq i \leq m-1)$ is a binary random variable. We have

$$E[Y_i] = 0 \cdot Pr(Y_i = 0) + 1 \cdot Pr(Y_i = 1) = Pr(Y_i = 1).$$

The AEM of the approximate circuit can be calculated as

$$E[Y] = E\left[\sum_{i=0}^{m-1} 2^i Y_i\right] = \sum_{i=0}^{m-1} 2^i E[Y_i] = \sum_{i=0}^{m-1} 2^i Pr(Y_i = 1).$$

Thus, to calculate AEM, we only need to obtain $Pr(Y_i = 1)$ for $0 \leq i < m$. Under a general input distribution, we can apply Monte Carlo-based logic simulation to obtain $Pr(Y_i = 1)$ approximately. If the input distribution is uniform, then we can calculate $Pr(Y_i = 1)$ *accurately* through BDD. Specifically, we first build the BDD for Y_i based on the circuit shown in Fig. 3. We then count the number of paths from the root of the BDD to its 1 terminal to get the total number of PI combinations that let Y_i take the value 1. This number divided by 2^n, where n is the number of inputs of the circuit, gives the probability $Pr(Y_i = 1)$ under the uniform input distribution. Note that counting the number of PI combinations that make Y_i take the value 1 is also known as the *model counting* or *#SAT* problem [36], [37]. Thus, alternatively, we can apply the solutions to such a problem to calculate AEM. In our experiments, we used the BDD-based method.

IV. Algorithms for Error Rate Constraint Only

In this section, we present two ALS algorithms for ER constraint only: ALFANS-ER and ALFANS-ER-Fast. ALFANS-ER picks the single most effective node to apply the approximate simplification at each iteration. However, it has long runtime. To address this drawback, we further propose ALFANS-ER-Fast. It picks multiple nodes to apply the approximate simplification simultaneously in each iteration.

A. Algorithm ALFANS-ER

The flow of algorithm ALFANS-ER is shown in Algorithm 1. It takes a network C and a given ER threshold D_{er} as inputs. We denote the approximate network as C', which is initially set as the input network C. The variable e records the ER of the network C' at the start of each iteration. Another variable d records the margin of the ER threshold, which equals $D_{er} - e$.

Algorithm 1: The ALS algorithm ALFANS-ER for ER constraint only.

```
1  input: the input network C and the ER threshold D_er;
2  output: an approximate network C' with ER ≤ D_er;
3  initialize: C' ⇐ C; current ER e ⇐ 0; ER margin d ⇐ D_er;
4  while true do
5      maxScore ⇐ −1;
6      for each node n in the network C' do
7          estimate the RERs of all ASEs for the node n;
8          bestASE ⇐ the ASE of n with the highest score and
             the RER ≤ d;
9          if bestASE.score > maxScore then
10             maxScore ⇐ bestASE.score;
11             maxScoreNode ⇐ n;
12     if maxScore < 0 then break;
13     for the maxScoreNode in C', replace its original factored
         expression by its bestASE;
14     update e; d ⇐ D_er − e;
15 return C';
```

The algorithm is iterative. In each iteration, it picks the single most effective node to apply the approximate simplification. Specifically, in each iteration, for each node n in the network C', Line 7 estimates the RERs of all the ASEs for the node n by the method described in Section III-D. Note that the RERs are defined over the approximate network C' in the current iteration. For a node n whose factored expression has N literals, there are $\binom{N}{1} + \binom{N}{2} + \cdots + \binom{N}{N-1} + 2\binom{N}{N} = 2^N$ different ASEs in total. We estimate the RERs for all these ASEs when $N < 5$. When $N \geq 5$, we only consider the ASEs obtained by removing fewer than 5 literals from the original expression as well as the constant-0 and constant-1 ASEs. Then, among all the *feasible* ASEs of the node n, Line 8 picks the one with the highest score as its best ASE. An ASE is defined to be *feasible* if its estimated RER, which gives an upper bound of the exact RER, is no more than the ER margin d. The score of an ASE is defined as l/er, where l is the reduced literal number over the original expression and er is the RER estimate of the ASE. The intuition behind the score function is that we want to select an ASE that maximizes the literal saving per unit of ER introduced.

We maintain two variables $maxScore$ and $maxScoreNode$ to record the highest score among all the ASEs over all the nodes and the node that gives $maxScore$, respectively (see Lines 10–11). After all the nodes are checked, if $maxScore$ is its initial value -1, it indicates that there are no feasible ASEs for the current approximate network and the while loop terminates (see Line 12), followed by returning the latest network C' as the final result (see Line 15). Otherwise, Line 13 replaces the original expression of node $maxScoreNode$ by its best ASE. Then, Line 14 calculates the actual ER e of the current network C' and updates the error margin d. The actual ER could be calculated through BDD if the input distribution is uniform or through logic simulation otherwise.

B. Algorithm ALFANS-ER-Fast

ALFANS-ER only selects one node to apply the approximate simplification in each iteration. Thus, it is not efficient when the number of iterations is large. To speed up the whole process, we propose an accelerated version, ALFANS-ER-Fast, where multiple nodes are picked and simplified in each iteration. There are two key questions related to this procedure: 1) How should we estimate the RER caused by the change of multiple nodes? 2) Which set of nodes and their associated ASEs should we pick to maximize the literal saving? We first answer these two questions in Section IV-B1 and Section IV-B2, respectively. Then, we show the overall flow of ALFANS-ER-Fast in Section IV-B3.

1) Estimating Real Error Rate Caused by Multiple Changes: When we select a set of nodes and their ASEs to replace their original expressions, we need to evaluate the RER caused by the functional changes of these nodes. However, we cannot calculate it simply by adding up the RERs of the ASEs for all the nodes involved in the change. The reason is that the RER of each ASE is calculated under the assumption that all the other nodes do not change. However, the change of some nodes after a node n in the topological order may alter the ODCs of n, which in turn changes the RER of an ASE for n. For example, consider simplifying two nodes n_1 and n_2 simultaneously, where n_1 is an immediate input of n_2. The RER of an ASE g for n_1 is calculated by assuming n_2 does not change. In our estimation of the RER of g, we ignore the ODCs of n_1 by setting their probabilities to 0. Now, due to the change of node n_2, some ODCs of n_1 may not be ODCs any more and hence, the RER of g may increase. On the other hand, if we consider the AER of an ASE for n_1, since its calculation is irrelevant to the nodes in the fanout cone of n_1, its value after the change of n_2 is still the same as before. Based on this observation, we propose to bound the RER caused by the multiple changes by the AERs of the ASEs for all the nodes involved in the change. Indeed, we have the following theorem.

Theorem 1 *Suppose q nodes $n_1, \ldots, n_q$ are picked to make simultaneous change. Assume the original expression of node n_i ($1 \leq i \leq q$) is replaced by an ASE g_i and the AER of g_i is r_i. Then the error rate introduced to the network by these changes is no more than $r_1 + r_2 + \cdots + r_q$.*

Due to the space limit, we show a sketch of the proof here. Define S_i as the set of AEPIPs of the ASE g_i. Let $p(V)$ denote the probability of the PI pattern V. Consider the new circuit obtained after replacing the original expression of node n_i by the ASE g_i for all $1 \leq i \leq q$. We first study which PI patterns could cause an error for the vector $(n_1, \ldots, n_q)$ in the new circuit. Suppose that these patterns form a set S. By mathematical induction on the nodes $n_1, \ldots, n_q$ in a topological order, we can prove that $S = S_1 \cup \cdots \cup S_q$. Therefore, the probability that the vector $(n_1, \ldots, n_q)$ is incorrect in the new circuit is

$$p_e = \sum_{V \in S_1 \cup \cdots \cup S_q} p(V) \leq \sum_{i=1}^{q} \sum_{V \in S_i} p(V).$$

By the definition of AER, we have $r_i = \sum_{V \in S_i} p(V)$. Thus, we have $p_e \leq \sum_{i=1}^{q} r_i$. Since the new circuit has some erroneous POs only if the vector $(n_1, \ldots, n_q)$ is incorrect, the ER of the new circuit is no more than p_e, which is further bounded by $\sum_{i=1}^{q} r_i$. This concludes the proof.

Based on the above theorem, in the multi-selection algorithm, we use the sum of the AERs of the ASEs for all the nodes involved in the change to estimate the ER increase of the new network in each iteration.

2) Selecting an Optimal Set of Nodes and their ASEs: In ALFANS-ER-Fast, we need to decide which set of nodes we should pick and among all the ASEs of each selected node, which one we should choose to maximize the literal saving.

We propose to formulate the above problem as a variation of the basic 0/1 knapsack problem, which we call *0/1 multi-state knapsack problem*. This problem is different from the basic knapsack problem in that each candidate item c_i has m_i states $(w_{i1}, v_{i1}), (w_{i2}, v_{i2}), \ldots, (w_{im_i}, v_{im_i})$, where w_{ij} and v_{ij} ($1 \leq j \leq m_i$) are the weight and the value, respectively, of the j-th state of the item c_i. We need to choose a set of items and their associated states to put into a knapsack with capacity W to maximize the total value.

The problem of finding an optimal set of nodes and their associated ASEs can be formulated as a 0/1 multi-state knapsack problem. Specifically, we treat each node as an item, each ASE of a node as a state of an item, the number of saved literals of an ASE as the value, the AER of an ASE as the weight, and the current ER margin d as the capacity of the knapsack.

To the best of our knowledge, the solution to this variation of the knapsack problem is unknown. We further propose a solution to this problem, which is a modification of the classical dynamic programming-based solution of the basic 0/1 knapsack problem.

The solution begins by filtering the states and items. We delete all the states with weight larger than the capacity W.

If all states of a candidate item are removed, then the item is also removed. For each remaining item, we further remove its remaining states that are dominated by another state. For two states $s_1 = (w_{i1}, v_{i1})$ and $s_2 = (w_{i2}, v_{i2})$ of an item c_i, state s_1 *dominates* state s_2 if and only if $w_{i1} \leq w_{i2}$ and $v_{i1} \geq v_{i2}$. In this case, if the item c_i is finally picked in the optimal solution, its state s_2 can never be the associated state. Thus, we remove the dominated state s_2.

For the remaining items and their states, we apply a dynamic programming-based method to find the optimal solution. The method is based on the classical solution to the basic 0/1 knapsack problem, which requires the weights to be non-negative integers. In our case, the weights, which correspond to the AERs, are non-negative real values. Therefore, we need to convert the AERs of all ASEs and the ER margin from reals to integers. Specifically, when the given ER threshold is less than 0.01, the ERs are multiplied by 10000. Otherwise, they are multiplied by 1000. Then, they are rounded to integers.

We use an example to further illustrate our proposed solution. Suppose there are 3 candidate items c_1, c_2, and c_3 and they have 2, 2, and 1 states, respectively, as shown in Table I. Assume the capacity of the knapsack is 9. To solve this multi-state knapsack problem, we define $m[i, j]$ to be the maximum value that can be attained with weight less than or equal to j using up to the first i items. We use a table shown in Table II to store the result of $m[i, j]$. The solution fills the table row by row from top to bottom. Since $m[0, j] = 0$, the first row corresponding to $i = 0$ has all the entries as 0.

In calculating $m[i, j]$ for $i \geq 1$, we need to consider all the states of the item c_i with weight less than or equal to j. For example, in calculating $m[2, 8]$, we need to consider both the states s_{21} and s_{22} of the item c_2. If we select c_2 and pick its state s_{21}, then the maximum value is $m[1, 8 - 4] + 2 = 2 + 2 = 4$. If we select c_2 and pick its state s_{22}, then the maximum value is $m[1, 8 - 6] + 4 = 1 + 4 = 5$. We should also consider the case where the item c_2 is not selected. In this case, the maximum value is $m[1, 8] = 2$. Comparing the above three choices, we can see that selecting c_2 and its state s_{22} gives the maximum possible value with weight less than or equal to 8 using up to the first 2 items. The maximum value is $m[2, 8] = 5$. Once we have constructed the entire table, the lower right corner of the table gives the optimal value of the problem. The final selected items and their states can be backtracked from the optimal value. For our example, the optimal value is 6 and the optimal choice consists of item c_1 with its state as s_{12} and item c_2 with its state as s_{22}.

TABLE I
CANDIDATE ITEMS AND THEIR STATES.

item	state	weight	value
c_1	s_{11}	2	1
	s_{12}	3	2
c_2	s_{21}	4	2
	s_{22}	6	4
c_3	s_{31}	2	1

3) Flow of ALFANS-ER-Fast: In this section, we present the flow of ALFANS-ER-Fast. It is shown in Algorithm 2. The definitions of the variables are the same as those in Algorithm 1. Lines 6–7 iterate over all nodes in network C' and put all ASEs of each node in the set S_{ASE}. Line 8 applies $Knapsack\_Solver$, which implements the solution to the knapsack problem discussed above, on the ASE set S_{ASE} and the error margin d to choose the optimal set of nodes and their associated ASEs. The selected nodes and their ASEs are stored in the set S_{select}. If S_{select} is empty, the loop terminates (see Line 9) and returns the latest approximate circuit C'. Otherwise, Lines 10–11 simplify the network C' based on the items in S_{select}. We then compute the actual ER of the network and update the ER margin d (see Line 12).

TABLE II
SOLUTION OF THE MULTI-STATE KNAPSACK PROBLEM SHOWN IN TABLE I.

	weight capacity									
up to item	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	2	2	2	2	2	2	2
2	0	0	1	2	2	2	4	4	5	6
3	0	0	1	2	2	3	4	4	5	6

Algorithm 2: The ALS algorithm ALFANS-ER-Fast for ER constraint only.

```
1  input: the input network C and the ER threshold D_er;
2  output: an approximate network C' with ER ≤ D_er;
3  initialize: C' ⇐ C; current ER e ⇐ 0; ER margin d ⇐ D_er;
4  while true do
5      S_ASE ⇐ ∅;
6      for each node n in the network C' do
7          find all ASEs of n and add them into S_ASE;
8      S_select ⇐ Knapsack_Solver(S_ASE, d);
9      if S_select = ∅ then break;
10     for each node n in S_select do
11         use the selected ASE to replace the original
             expression of n;
12     update e; d ⇐ D_er − e;
13 return C';
```

V. ALGORITHM FOR COMBINED CONSTRAINTS ON ERROR RATE AND MAXIMUM ERROR MAGNITUDE

In this section, we present ALFANS-ER-MEM, the ALS algorithm for the combined constraints on ER and MEM. Same as ALFANS-ER described in Section IV-A, it is an iterative algorithm and in each iteration, the best ASE is picked to simplify the current approximate network. An important component of the algorithm is the scoring mechanism, which we will introduce in Section V-A. Then, we will describe the proposed algorithm in Section V-B.

A. Scoring Mechanism

With the extra MEM bound, we need to ensure that after applying a simplification, the MEM of the approximate network is still within the bound. To guarantee this, for each ASE with RER within the current ER margin, we check the MEM of the network using the technique introduced in Section III-E. If the MEM is below the bound, we call the ASE an *ER-MEM-feasible ASE*. Similar to the case where the constraint only includes ER, a network usually has multiple ER-MEM-feasible ASEs in each iteration. ALFANS-ER-MEM picks the best ER-MEM-feasible ASE to perform simplification in each iteration. One important problem is to define a good scoring mechanism to measure the quality of an ASE.

In ALFANS-ER, the score of an ASE is defined as l/er, where l is the number of saved literals and er is the RER of an ASE. This score definition only considers the impact of an ASE on ER, but not the impact on MEM. Our experiments showed that using this score definition for the combined

constraints on ER and MEM does not perform well on some benchmarks. Our analysis reveals that this is because the algorithm will choose ASEs that increase the MEM of the network a lot in several initial iterations and thus, will exhaust all the space for further simplification, even if the ER of the approximate network is still far less than the ER threshold.

Thus, we redesign the score function of an ASE by involving both of the ASE's impacts on ER and MEM as follows:

$$s = \frac{l}{s_{er} s_{mem}}, \tag{1}$$

where l is the number of saved literals and s_{er} and s_{mem} represent the impacts of the ASE on ER and MEM, respectively. We describe how s_{er} and s_{mem} are defined in the next two subsections.

1) Measure for the Impact of an ASE on Error Rate: s_{er} is defined based on the ER of the entire network. Denote the approximate network in the previous iteration as C_{pre}. In the current iteration, we need to evaluate the impact on ER for each ASE derived from the network C_{pre}. To obtain the ER impact of a specific ASE g, we need to evaluate the ER of the new network C_{new} obtained by applying g to the approximate network C_{pre}. If we use logic simulation to do so, we need to repeat it for each ASE. This is time-consuming, since there are generally many ASEs for a network. To improve the efficiency, we propose to estimate the ER of the new network C_{new} from the ER of the previous network C_{pre} and the RER of ASE g.

Denote the set of PI patterns that causes the output error for network C_{pre} as S_1, the set of REPIPs of ASE g as S_2, and the set of PI patterns that causes the output error for network C_{new} as S_3. Next, we will derive a relation among sets S_1, S_2, S_3.

Note that a pattern in S_1 causes output difference between C_{pre} and the original network, while a pattern in S_2 causes output difference between C_{new} and C_{pre}. We can partition the set of all PI patterns into four subsets: $S_2 \cap S_1$, $S_2 \cap \overline{S_1}$, $\overline{S_2} \cap S_1$, and $\overline{S_2} \cap \overline{S_1}$. By definition, a pattern in set $S_2 \cap \overline{S_1}$ or set $\overline{S_2} \cap S_1$ causes output difference between network C_{new} and the original network, while a pattern in set $\overline{S_2} \cap \overline{S_1}$ does not. A pattern in set $S_2 \cap S_1$ causes output difference between C_{new} and C_{pre} and output difference between C_{pre} and the original network. For a multi-output network, it is unlikely that the first difference exactly cancels the second. Thus, a pattern in set $S_2 \cap S_1$ also causes output difference between network C_{new} and the original network. Thus, we have

$$S_3 = (S_2 \cap S_1) \cup (S_2 \cap \overline{S_1}) \cup (\overline{S_2} \cap S_1) = S_1 \cup S_2.$$

Thus, the ER e_{new} of network C_{new} can be calculated as

$$e_{new} = Pr(S_1) + Pr(S_2) - Pr(S_1 \cap S_2), \tag{2}$$

where $Pr(S)$ denotes the probability of a PI pattern occurring in set S. Denote the ER of network C_{pre} as e_{pre} and the RER of ASE g as e_{ASE}. By definition, we have $Pr(S_1) = e_{pre}$ and $Pr(S_2) = e_{ASE}$. Assuming that the occurrence of a PI pattern in set S_1 is independent to its occurrence in set S_2, we have $Pr(S_1 \cap S_2) = Pr(S_1)Pr(S_2) = e_{pre}e_{ASE}$. Thus, we can simplify Eq. (2) as follows:

$$e_{new} = e_{pre} + e_{ASE} - e_{pre}e_{ASE}. \tag{3}$$

The above equation gives a way to estimate the ER of the new network C_{new} from the ER of the previous network C_{pre} and the RER of ASE g.

Our proposed measure for the impact of an ASE on ER is calculated based on e_{new} as follows:

$$s_{er} = \frac{e_{new}}{D_{er} - e_{new}}, \tag{4}$$

where D_{er} is the given ER threshold. We do not directly use e_{new} as a measure of the ER impact, since we find it is beneficial to enlarge the difference between the ER impacts of two ASEs with different e_{new}'s. One way to achieve this is dividing e_{new} by $D_{er} - e_{new}$, as Eq. (4) shows.

2) Measure for the Impact of an ASE on Maximum Error Magnitude: To obtain the exact MEM caused by an ASE essentially needs to enumerate all input patterns, which could be very time-consuming. In this section, we show an efficient empirical measure s_{mem}.

First, we observe that for an arbitrary node n in a network, its transitive fanout cone usually only contains a subset of all POs. For a node n, we call a PO in the transitive fanout cone of n a *reachable PO* of n. The simplification of n with its ASE can only affect the values of its reachable POs. Assume the input network has m POs $O_0, O_1, \cdots, O_{m-1}$ and that the weight of PO O_i $(0 \leq i < m)$ in the encoded numerical value is 2^i. Then, for a node n with j reachable POs $O_{i_1}, O_{i_2}, \cdots, O_{i_j}$ $(0 \leq i_1 < \cdots < i_j < m)$, the MEM over the previous approximate network caused by an ASE of n is at least $M_{min} = 2^{i_1}$ and at most $M_{max} = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_j}$. For simplicity, we choose to define an ASE's impact on the MEM of the entire approximate network, s_{mem}, as a function on either M_{min} or M_{max}. Our experimental study shows that choosing M_{min} as an indicator of an ASE's impact on the total MEM gives smaller final circuit area. For example, for benchmark CLA16 in our experiments, if we chose M_{min} as the indicator, the areas of the final approximate circuits under ER threshold and MEM threshold pairs $(D_{er}, D_{mem}) = (60\%, 300)$ and $(70\%, 300)$ are 251 and 233, respectively. However, if we chose M_{max} as the indicator and kept all the other parts of the algorithm unchanged, the final areas grow to 264 and 256, respectively. This is because neither of M_{min} and M_{max} equals the actual MEM. The former is an underestimate of the actual MEM, while the latter is an overestimate. In this case, using an underestimate helps check more candidate choices and eventually leads to a better final result. Therefore, we choose M_{min} as an indicator of an ASE's impact on the total MEM. The actual function for s_{mem} is defined as follows:

$$s_{mem} = e^{(\log_2 M_{min})/T} = e^{i_{min}/T}, \tag{5}$$

where $i_{min} = i_1$, the index of the reachable PO of n with the minimum weight, and T is a positive control parameter similar to the temperature in *simulated annealing* [38]. Its function is to dynamically adjust the relative differences of the s_{mem}'s among ASEs with different i_{min}'s. The parameter T begins with an initial value T_b and gradually decreases over iterations until it becomes less than a final threshold T_f. We let T decrease over iterations because we find that if s_{mem}'s for ASEs with different i_{min}'s are close at first and then become more and more different later on, the qualities of the finally synthesized approximate circuits are better than other settings. By Eq. (5), a large T causes s_{mem}'s for ASEs with different i_{min}'s to be close, while a small T causes them to be different. Thus, we let T decrease over iterations.

The criterion that guides the change of T is as follows. For a node n, define i_{max} as the index of the reachable PO of n with the maximum weight. After each iteration, the value of T is updated as

$$T = T_{pre} \cdot (\alpha D_{er})^{\log_2(1+NC)}, \tag{6}$$

where T_{pre} is the value of T in the previous iteration, $0 < \alpha < 1$ is a constant parameter, D_{er} is the ER threshold, and

the value NC is the number of consecutive previous iterations in which the picked nodes have their $2^{i_{max}} \leq D_{mem}$, where D_{mem} is the MEM threshold. Note that a special case is that if $2^{i_{max}} > D_{mem}$ for the picked node in the previous iteration, then $NC = 0$ and thus, T is unchanged.

In the above update scheme of T, we use αD_{er} as the base of the decreasing ratio. This is because a larger ER threshold D_{er} typically requires more iterations. Thus, for a larger D_{er}, due to more number of iterations, T should decrease more slowly. Using αD_{er} as the base realizes the above relation between the decreasing ratio and the value of D_{er}. In our experiments, constant α is set as 0.8.

Another problem is how to determine the initial value of T, T_b, and the final stopping threshold of T, T_f. They depend on the input circuit and the thresholds D_{er} and D_{mem}. It is hard to decide the best choices for T_b and T_f beforehand. Therefore, we assign several (T_b, T_f) pairs to run the algorithm and finally select the best approximate circuit. In our study, we find that that a critical value for T_b is 30. T_b larger than 30 usually produces worse results than T_b less than or equal to 30 for most benchmarks in our experiments. Besides, when the ratio T_b/T_f is in the range $[10, 20]$, it is more prone to produce approximate circuits with less area. Therefore, we chose to run the experiments for all benchmarks with five different (T_b, T_f) pairs, $(5, 0.5)$, $(15, 1)$, $(20, 1)$, $(20, 2)$, and $(30, 3)$.

B. Flow of ALFANS-ER-MEM

The flow of ALFANS-ER-MEM is shown in Algorithm 3. It takes as inputs the input network C, the ER threshold D_{er}, the MEM threshold D_{mem}, the initial value of the control parameter T, T_b, and the final stopping threshold for T, T_f.

Algorithm 3: The ALS algorithm ALFANS-ER-MEM for the combined constraints on ER and MEM.

```
1  input: the input network C, the ER threshold D_er, the MEM
     threshold D_mem, the initial value of T, T_b, and the final
     stopping threshold for T, T_f;
2  output: an approximate network C' with ER ≤ D_er and MEM
     ≤ D_mem;
3  initialize: C' ⇐ C; current ER e ⇐ 0; T ⇐ T_b;
4  while true do
5      maxScore ⇐ −1; S_cand ⇐ ∅; picked ⇐ false;
6      for each node n in the network C' do
7          for each ASE g of node n do
8              e_g ⇐ the estimated RER of g;
9              if e + e_g − e · e_g ≤ D_er then
10                 g.score ⇐ the score of g calculated by
                     Eq. (1) with parameter T;
11                 if g.score ≥ maxScore then
12                     if the MEM of the resulting network after
                         applying g to C' is within D_mem then
13                         add g into S_cand;
14                         maxScore ⇐ g.score;
15     for each ASE g in S_cand with score in descending order do
16         obtain the accurate ER e_acc of the resulting network
             after applying g to C';
17         if e_acc ≤ D_er then
18             apply g to simplify C'; e ⇐ e_acc;
19             picked ⇐ true;
20             break;
21     if picked = false then break;
22     update T by Eq. (6);
23     if T < T_f then T ⇐ T_f;
24 return C';
```

The algorithm is iterative. In each iteration, we first obtain a set S_{cand} of candidate ASEs (see Lines 6–14) and then pick the best ASE from S_{cand} (see Lines 15–20). We use a variable $picked$ to indicate whether in the current iteration, an ER-MEM-feasible ASE is successfully picked. It is initialized to false at the beginning of each iteration (see Line 5).

There are many ASEs for a network. An ER-MEM-feasible ASE should satisfy that after applying it to the current network, the MEM of the network is below the given MEM threshold. The check of this condition, which is based on the technique described in Section III-E, is time-consuming. To reduce the number of checks, we first filter all ASEs by the ER constraint and the scores of the ASEs. To achieve this, Line 8 first estimates the RER e_g of the ASE g by the technique described in Section III-D. Then, we apply Eq. (3) to estimate the ER of the new network obtained by applying g to the current network C'. Note that e_{pre} and e_{ASE} in Eq. (3) are replaced by e and e_g, respectively. Thus, the ER of the new network is $e_{new} = e + e_g - e \cdot e_g$. If $e_{new} \leq D_{er}$, we further compute the score of ASE g by Eq. (1) (see Lines 9–10). We will only do the MEM check if the score of g is no less than the maintained maximum score $maxScore$ (see Lines 11–14). This saves the runtime by avoid checking the MEM of many low-score ASEs. If g does not cause the MEM of the current network C' to exceed D_{mem}, we add g into S_{cand} and update $maxScore$ (see Lines 12–14).

After the set S_{cand} of candidate ASEs is obtained, we then pick the best ASE from S_{cand}. Note that the ERs of the networks simplified by the ASEs in S_{cand} are estimated by Eq. (3). Thus, they are not exact and may actually exceed the threshold D_{er}. In this step, we obtain the accurate ER of each network simplified by an ASE in S_{cand} (see Line 16). The accurate ER could be calculated through BDD if the input distribution is uniform or through logic simulation otherwise. Since the accurate ER calculation is also time-consuming, we propose to visit the ASEs in descending order of their scores (see Line 15). As soon as we have identified an ASE such that applying it to the current network C' does not cause the ER of the new network to exceed D_{er}, we pick this ASE to simplify C' and stops checking the remaining ASEs in S_{cand} (see Lines 17–20). In this case, the ER e of the current network is updated to the accurate ER of the new network. Also, the variable $picked$, which records whether an ER-MEM-feasible ASE is successfully picked, is set as true (see Line 19). After all the ASEs in S_{cand} are checked, if $picked$ is still false, then it implies that no ER-MEM-feasible ASE is found in S_{cand} and the outer loop terminates (see Line 21). Otherwise, we update T by Eq. (6) and begins the next iteration. If T drops below the threshold T_f, we set it as T_f.

VI. Algorithm for Combined Constraints on Error Rate and Average Error Magnitude

In this section, we present ALFANS-ER-AEM, the ALS algorithm for the combined constraints on ER and AEM. This algorithm still sticks to the framework of ALFANS-ER described in Section IV-A. We call an ASE an *ER-AEM-feasible* ASE if applying it to the current network does not cause the ER and the AEM of the network to exceed their corresponding bounds. Our strategy is to pick the best ER-AEM-feasible ASE to perform simplification in each iteration. Different from the situation where the MEM is part of the constraints, when the constraints include AEM, we can leverage the technique introduced in Section III-F to calculate the accurate AEM of a network. This influences the scoring mechanism, which we will describe next.

The proposed score function for an ASE under the combined constraints on ER and AEM is as follows:

$$s = \frac{l}{s_{er} s_{aem}}, \tag{7}$$

where l is the literal save of the ASE and s_{er} and s_{aem} represent the impacts of the ASE on the ER and AEM, respectively. This function is similar to the one shown in Eq. (1). Besides, the calculation of s_{er} is same as Eq. (4). However, the function for s_{aem} is different from that for s_{mem}. It is defined as follows:

$$s_{aem} = e^{\frac{\log_2(V_{aem})}{T}}, \tag{8}$$

where V_{aem} is the AEM of the resulting network after the ASE is applied to the current network.

The difference between the formulas of s_{aem} and s_{mem} lies in that we use $\log_2(V_{aem})$ to replace $\log_2(M_{min})$ in Eq. (5). The reason for this change is because when the constraints include MEM, we cannot very efficiently obtain the exact error magnitude that an ASE can add to the MEM of the entire network. Instead, we find that M_{min} is a good indicator of an ASE's impact on the total MEM. Thus, we define s_{mem} as a function of M_{min}. However, since the constraint on error magnitude has shifted from MEM to AEM and we can get the exact AEM of an approximate network by the technique described in Section III-F, it is more accurate now to use V_{aem} in the s_{aem} formula to reflect an ASE's impact on the total AEM. This explains why we use $\log_2(V_{aem})$ to replace $\log_2(M_{min})$ in Eq. (5) to derive the formula for s_{aem}.

The variable T in the formula for s_{aem} has the same meaning as the variable T in the formula for s_{mem}. In addition, its updating formula is same as Eq. (6). The only change is to the variable NC in Eq. (6). Now, NC is defined as the number of consecutive previous iterations such that the AEM of the approximate network in one iteration increases over the previous iteration.

The flow of ALFANS-ER-AEM is shown in Algorithm 4. Its inputs are similar to those of Algorithm 3, except that the MEM threshold D_{mem} in Algorithm 3 is replaced by the AEM threshold D_{aem}. During the iterations, we use $V_{aem,c}$ to represent the AEM of the current approximate network C'.

Although the accurate AEM of an approximate network can be calculated by the technique described in Section III-F, it is time-consuming to evaluate the AEMs of all approximate networks corresponding to all ASEs, because a circuit has many candidate ASEs and the evaluation of the AEM for each ASE takes non-negligible time no matter using the logic simulation-based method, BDD-based method, or the method to solve the model counting problem. Thus, the flow first excludes some unpromising ASEs from checking (see Lines 8–14). For each ASE g of each node n in network C', Line 9 first applies Eq. (3) to estimate the ER e_{est} of the new network obtained by applying g to the current network C'. If it is smaller than the ER threshold D_{er}, we further check whether ASE g could lead to a higher score than the current maintained maximum score $maxScore$. This is achieved by doing a quick estimation of the upper bound of the score, which is shown in Lines 11–14. The logic behind this estimation is explained as follows.

By Eq. (7), the score of an ASE depends on s_{er}, which, by Eq. (4), further depends on the ER of the resulting approximate network after the ASE is applied to the current approximate network. However, the calculation of the accurate ER of the resulting approximate network is time-consuming. Thus, to get a faster estimation, we replace e_{new} in Eq. (4) by the estimated ER of the resulting approximate network, i.e., e_{est}. We have

$$s = \frac{l}{s_{er} s_{aem}} \approx \frac{l}{\frac{e_{est}}{D_{er} - e_{est}} \cdot s_{aem}}. \tag{9}$$

By Eq. (9), the score of an ASE depends on s_{aem}, which, by Eq. (8), further depends on the AEM of the resulting approximate network after the ASE is applied to the current approximate network. However, as we stated above, the calculation of the accurate AEMs for all candidate ASEs is time-consuming. Thus, we cannot afford the accurate AEM calculation for each ASE. As we observed, the AEM of an approximate network usually increases after the application of an ASE, but in occasional cases, it could also decrease because the introduced ASE cancels some existing errors. However, it almost always holds that the AEM of an approximate network after the application of an ASE is larger than half of the AEM of the current approximate network. Thus, as a conservative estimate, we have that the value s_{aem} of an ASE is at least $e^{\log_2(0.5V_{aem,c})/T}$, where $V_{aem,c}$ is the AEM of the current approximate network C'. Thus, the score for an ASE of network C' satisfies

$$s \approx \frac{l}{\frac{e_{est}}{D_{er} - e_{est}} \cdot s_{aem}} \leq \frac{l}{\frac{e_{est}}{D_{er} - e_{est}} \cdot e^{\log_2(0.5V_{aem,c})/T}}. \tag{10}$$

The right-hand side (RHS) of Eq. (10) gives an upper bound of the score of the ASE. If the upper bound is smaller than $maxScore$, we will not further consider this ASE.

In our actual realization shown in Lines 11–14, we distinguish based on whether $V_{aem,c} > 2$ or not. If $V_{aem,c} > 2$, then we calculated the upper bound by the RHS of Eq. (10).

Algorithm 4: The ALS algorithm ALFANS-ER-AEM for the combined constraints on ER and AEM.

```
1  input: the input network C, the ER threshold D_er, the AEM
     threshold D_aem, the initial value of T, T_b, and the final
     stopping threshold for T, T_f;
2  output: an approximate network C' with ER ≤ D_er and AEM
     ≤ D_aem;
3  initialize: C' ⇐ C; current ER e ⇐ 0; current AEM
     V_aem,c ⇐ 0; T ⇐ T_b;
4  while true do
5      maxScore ⇐ −1;
6      for each node n in the network C' do
7          for each ASE g of node n do
8              e_g ⇐ the estimated RER of g;
9              e_est = e + e_g − e · e_g;
10             if e_est ≤ D_er then
11                 l ⇐ the number of saved literals of g;
12                 g's partial score s_est = l/(e_est/(D_er − e_est));
13                 if V_aem,c > 2 then
14                     s_est = s_est/e^{log2(0.5 V_aem,c)/T};
15                 if s_est ≥ maxScore then
16                     obtain the accurate AEM V_acc and ER
                         e_acc of the resulting network after
                         applying g to C';
17                     if V_acc ≤ D_aem and e_acc ≤ D_er then
18                         s ⇐ l/(e_acc/(D_er − e_acc) · e^{log2(V_acc)/T});
19                         if s > maxScore then
20                             maxScore ⇐ s; bestASE ⇐ g;
21     if maxScore < 0 then break;
22     simplify C' by bestASE;
23     obtain the ER e and AEM V_aem,c of C';
24     update T by Eq. (6);
25     if T < T_f then T ⇐ T_f;
26 return C';
```

Otherwise, we set the term $e^{\log_2(0.5V_{aem,c})/T}$ in the RHS of Eq. (10) to 1 and calculate the upper bound as $l/(\frac{e_{est}}{D_{er}-e_{est}})$. The reason for this is because the upper bound estimated by the RHS of Eq. (10) is not exact. We find that when $V_{aem,c} \leq 2$, the term $e^{\log_2(0.5V_{aem,c})/T}$ in the RHS of Eq. (10) is less than or equal to 1. In this case, the upper bound is too loose and we find it is beneficial to set the term to 1.

With the upper bound calculated, we will only formally check an ASE if the upper bound is larger than or equal to $maxScore$ (see Lines 15–20). In this case, we first obtain the accurate AEM and ER of the resulting network after the ASE is applied. If they are within the corresponding bounds, we calculate the score of the ASE by Eq. (7). If the score is larger than $maxScore$, we update $maxScore$ by the score of the ASE and set the best ASE $bestASE$ as the ASE. After we have iterated over all ASEs of all nodes in network C', if $maxScore$ is still its initial value -1, then it means that no ASE is selected in this iteration and we terminate the loop (see Line 21). Otherwise, Lines 22–23 simplify network C' by $bestASE$ and obtain the accurate ER and AEM of the new network. Then, Lines 24–25 update the parameter T in a similar way as it is done in Algorithm 3.

VII. Experimental Results

In this section, we present the experimental results of our proposed algorithms. We implemented each algorithm in C++ and integrated it into the SIS flow [39] to build an ALS flow. The ALS flow begins by reading the circuit and establishing the Boolean network. Then, our ALFANS algorithm is applied to simplify the Boolean network, followed by calling the sweep operation in SIS to propagate constant and eliminate dangling wires. Finally, the technology mapping operation in SIS is called to build the final approximate circuit. We tested our ALS flows on a set of benchmarks, including some circuits in the MCNC benchmark set [40] and some common arithmetic circuits. For the baseline accurate circuits, the circuits in the MCNC benchmark set were optimized by SIS, while the arithmetic circuits were optimized by Synopsys Design Compiler [41]. The MCNC generic standard cell library was used as the cell library. All the experiments were conducted on a virtual machine running Linux operating system with 1GB memory. The host machine is a 3.1GHz desktop.

In our experiments, we assumed that the inputs are uniformly distributed. The AER and RER of an ASE were estimated through logic simulation using the Verilog logic simulation tool Synopsys VCS [41]. Each run of logic simulation applied 10000 randomly generated PI vectors, which was able to provide accurate enough estimate of AERs and RERs. The SAT solver we used for checking the MEM against the threshold is Minisat [42]. When the accurate AEM and ER of an approximate circuit are needed in an algorithm, we obtained them by BDD using the CUDD package [43].

A. The Algorithms for Error Rate Constraint Only

In this section, we studied the performance of the two proposed algorithms for the ER constraint only, ALFANS-ER and ALFANS-ER-Fast.

For comparison, we also implemented the state-of-the-art approach SASIMI proposed in [17]. However, since our algorithms do not consider the timing constraints of the circuits, for fair comparison, we also excluded the operations that handle timing constraints in the implementation of SASIMI. In SASIMI, gate downsizing is applied to further reduce the area. For our approaches, since they work at the technology-independent synthesis phase, gate downsizing is not applicable. For fair comparison, we also excluded the gate downsizing operation in the implementation of SASIMI.

TABLE III
Benchmarks used for the algorithms for error rate constraint only.

circuit	#I/Os	function	#nodes	area	delay
c880	60/26	8-bit ALU	357	599	40.4
c1908	33/25	16-bit SEC/DED circuit	880	1013	60.6
c2670	233/140	12-bit ALU and controller	1153	1434	67.3
c3540	50/22	8-bit ALU	629	1615	84.5
c5315	178/123	9-bit ALU	893	2432	75.3
c7552	207/108	32-bit adder/comparator	1087	2759	159.8
alu4	14/8	ALU	730	2740	51.5
RCA32	64/33	32-bit ripple-carry adder	202	691	42.8
CLA32	64/33	32-bit carry-lookahead adder	303	1063	45.8
KSA32	64/33	32-bit Kogge-Stone adder	345	1128	27
MUL8	16/16	8-bit array multiplier	436	1276	67.9
WTM8	16/16	8-bit Wallace tree multiplier	382	1104	69.6

Table III lists the information of the benchmark circuits used in this set of experiments. The area and delay for each circuit are the results optimized by SIS or Design Compiler as mentioned above. For some benchmarks, there exist some initial redundancies: one node in the circuit has exactly the same global function as another node. Such redundancy cannot be identified and removed by the available logic synthesis tool. SASIMI, which performs pairwise comparison of the signals, is able to detect identical signal pairs while our approaches cannot. To enhance the optimization power of our approaches, we further applied a simple pre-process method to remove the redundancies in the input circuit before carrying out the proposed ALS algorithms. This method is based on the observation that two identical signals must have the same PI supports. In the method, we check each node in topological order and find if there are any other nodes that have the same PI supports as the current node. If yes, we then compare their signatures obtained from logic simulation. Two nodes with the same supports and the same signatures are treated as identical and the one that can cause more literal reduction is replaced by the other. This pre-process step efficiently removes the initial redundancies in the circuit. Its runtime was counted into the total runtime of our proposed algorithms.

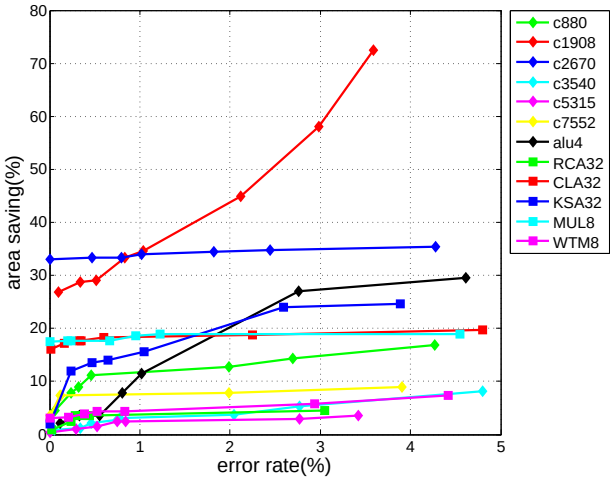


Fig. 4. Area saving versus error rate of ALFANS-ER.

Fig. 4 shows the area saving versus ER for all the benchmarks by applying ALFANS-ER. We can see that for each benchmark, as the ER increases, the area saving increases as well. When the ER threshold is 5%, for most benchmarks, their approximate circuits have $15\% \sim 35\%$ area saving. For c1908, the area saving even reaches 72%. Note that for benchmarks c2670, CLA32, and MUL8, when the ER is 0, the area saving is larger than 0. This is because ALFANS-ER is able to identify some redundancy which cannot be captured by the traditional logic synthesis tool. The plot of the area saving versus ER of ALFANS-ER-Fast is close to that of ALFANS-ER. Due to the space limit, we do not show it here.

Comparison in area saving and runtime among SASIMI, ALFANS-ER, and ALFANS-ER-Fast is presented in Table IV. For each benchmark, we ran the experiments for 7 different ER thresholds: 0.1%, 0.3%, 0.5%, 0.8%, 1%, 3%, and 5%. For each ER threshold, the ratio of the area of the obtained approximate circuit to that of the original circuit was calculated. Then, the average value of the 7 ratios was computed and listed in the "area ratio" columns in Table IV. To compare the runtime, we also computed the average runtime in seconds over the 7 experiments on different ER thresholds and listed it in the "time(s)" column.

TABLE IV
AREA RATIO, DELAY RATIO, AND RUNTIME COMPARISONS AMONG SASIMI [17], ALFANS-ER, AND ALFANS-ER-FAST.

	SASIMI			ALFANS-ER			ALFANS-ER-Fast		
circuit	area ratio	delay ratio	time (s)	area ratio	delay ratio	time (s)	area ratio	delay ratio	time (s)
c880	0.896	0.953	154	0.893	0.936	93	0.893	0.946	48
c1908	0.610	0.865	1090	0.595	0.850	394	0.598	0.868	181
c2670	0.724	0.682	664	0.662	0.626	702	0.673	0.692	90
c3540	0.975	0.991	393	0.966	0.985	172	0.965	0.989	77
c5315	0.981	0.989	996	0.978	0.989	263	0.981	0.989	85
c7552	0.948	0.977	2665	0.940	0.972	533	0.941	0.937	173
alu4	0.892	0.969	645	0.878	0.934	1000	0.869	0.932	186
RCA32	0.972	0.694	33	0.970	0.699	40	0.969	0.724	15
CLA32	0.829	1.251	196	0.822	1.275	213	0.822	1.275	57
KSA32	0.830	0.833	553	0.827	0.833	495	0.831	0.826	39
MUL8	0.829	0.925	1095	0.819	0.916	223	0.826	0.926	151
WTM8	0.959	0.937	249	0.953	0.897	168	0.956	0.940	57
Geomean	0.863	0.911	452	0.849	0.896	260	0.852	0.910	77

Comparing ALFANS-ER with ALFANS-ER-Fast, we can see that for most cases, ALFANS-ER produces circuits with slightly smaller area than ALFANS-ER-Fast, but the reverse is true for benchmarks c3540, alu4, and RCA32. On average, the area of the circuit obtained by ALFANS-ER-Fast is larger than that of the circuit obtained by ALFANS-ER by only 0.4% and the maximal area difference between the two algorithms over all benchmarks is 1.7%. Thus, we conclude that ALFANS-ER-Fast produces approximate circuits with almost the same area as ALFANS-ER. Note that the reason that for some cases ALFANS-ER is slightly better and for the other cases ALFANS-ER-Fast is slightly better is because both algorithms are inherently greedy heuristic algorithms and they cannot produce the exact optimal solutions. In terms of runtime efficiency, on average, ALFANS-ER-Fast has a speedup of $3.4\times$ over ALFANS-ER. Overall, ALFANS-ER-Fast achieves a significant speed-up over ALFANS-ER with almost the same circuit area. Hence, it may be a preferable choice when the error constraint only involves ER constraint.

Comparing the proposed algorithms ALFANS-ER and ALFANS-ER-Fast to SASIMI, we find that ALFANS-ER performs better in area saving than SASIMI for all the benchmarks and that ALFANS-ER-Fast performs better than SASIMI for all the benchmarks except KSA32. On average, ALFANS-ER and ALFANS-ER-Fast reduce the circuit area by 1.6% and 1.3%, respectively, than SASIMI. As shown in Table IV, the major advantage of our approaches lies in runtime. On average, the speedups of ALFANS-ER and ALFANS-ER-Fast over SASIMI are $1.7\times$ and $5.9\times$, respectively. SASIMI is slower than our approaches because it performs pairwise comparisons of all the signals in the network. Thus, its runtime complexity is quadratic to the number of nodes in the network. In contrast, both of our approaches have runtime complexity linear to the number of nodes in the network.

Although circuit delay is not the optimization target, we also present the effect of the three methods on delay in Table IV. For each benchmark, we show the average ratio of the delay of the obtained approximate circuit to that of the original circuit over the 7 ER thresholds in the "delay ratio" columns of Table IV. From the data in "delay ratio" columns, we can see that although the proposed approaches do not set timing constraint for the circuit, for all the benchmarks except CLA32, the obtained approximate circuits have smaller delays than the original circuits. This is because our proposed approximate simplification technique always changes a node in a Boolean network to a simpler form. Thus, each node in the Boolean network is either simplified or kept unchanged. Consequently, the final circuit produced after technology mapping usually will not have its delay increased. The delay increase of CLA32 is due to the initial pre-process step of removing redundancies. In this step, if we find two nodes with the same signature, we always replace the one that could have more literal reduction by the other. This step does not consider the impact on delay explicitly and causes the final delay increase for CLA32.

Comparing ALFANS-ER with SASIMI, we can see that for most benchmarks, the delays of the approximate circuits produced by ALFANS-ER are smaller than those of the approximate circuits produced by SASIMI. The geometric means of all the delay ratios for ALFANS-ER and SASIMI are 0.896 and 0.911, respectively, which demonstrates that ALFANS-ER is also better than SASIMI in delay optimization. Comparing ALFANS-ER-Fast with SASIMI, we can see that for nearly half of the benchmarks, the approximate circuits produced by ALFANS-ER-Fast have smaller delays than those produced by SASIMI. The geometric means of all the delay ratios for ALFANS-ER-Fast and SASIMI are 0.910 and 0.911, respectively, which shows that ALFANS-ER-Fast and SASIMI have very similar effects on delay optimization.

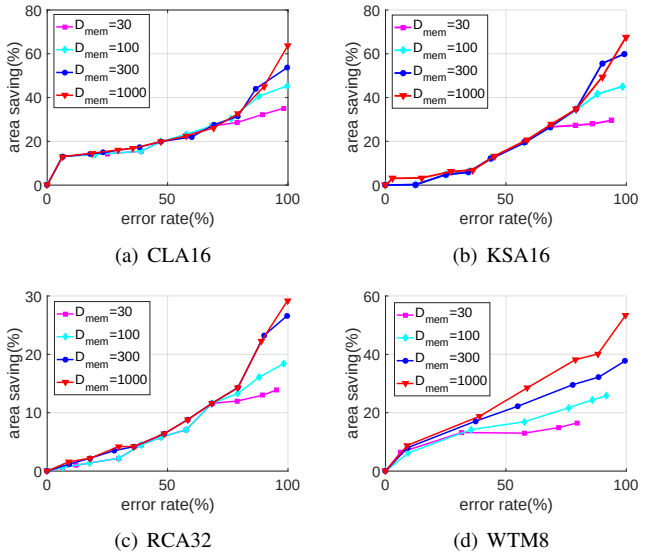
B. The Algorithm for the Combined Constraints on Error Rate and Maximum Error Magnitude

In this section, we studied the performance of the proposed ALS algorithm ALFANS-ER-MEM for the combined constraints on ER and MEM. We compared our algorithm to that proposed in [19], which is the state-of-the-art method for the same kind of combined error constraints. Since the MEM constraint is usually meaningful for arithmetic circuits, we chose the arithmetic circuits listed in Table V as the benchmarks. They were also used in the work [19] as the benchmarks. These circuits were pre-processed as introduced in Section VII-A before they were fed into the algorithms.

We tested each benchmark on four different MEM thresholds D_{mem}: 30, 100, 300, and 1000. For each D_{mem}, the ER threshold D_{er} was varied from 10% to 100%. For each constraint pair (D_{er}, D_{mem}) of each benchmark, we ran Algorithm 3 with 5 different (T_b, T_f) pairs, which are

TABLE V
BENCHMARKS USED FOR THE ALGORITHMS FOR THE COMBINED ERROR CONSTRAINTS.

circuit	#I/Os	function	#nodes	area	delay
CLA16	32/17	16-bit carry-lookahead adder	79	322	45.1
KSA16	32/17	16-bit Kogge-Stone adder	137	421	24.1
RCA32	64/33	32-bit ripple-carry adder	202	691	42.8
WTM8	16/16	8-bit Wallace tree multiplier	382	1104	69.6



(a) CLA16 (b) KSA16

(c) RCA32 (d) WTM8

Fig. 5. Area saving versus error rate of the final approximate circuit by ALFANS-ER-MEM for different benchmarks and different maximum error magnitudes.

$(5, 0.5)$, $(15, 1)$, $(20, 1)$, $(20, 2)$, and $(30, 3)$, and chose the final approximate circuit as the one with the smallest area among the five resulting circuits. The area saving in percentage versus the ER of the final approximate circuits for the four benchmark circuits is plotted in Fig. 5. As the figure shows, the area saving increases monotonically with D_{er} for all the benchmarks when D_{mem} is fixed. Moreover, when D_{er} is fixed, the area saving usually also increases with D_{mem}. This is reasonable since a larger D_{mem} implies a larger space for approximation. However, the difference of the area saving for different D_{mem}'s is usually not obvious when D_{er} is small and the difference grows as D_{er} approaches 100%. For example, for CLA16, when $D_{er} < 0.8$, the area saving of the final approximate circuits keeps almost unchanged when D_{mem} increases from 30 to 1000. The reason for this is because the constraints of ER and MEM are not completely independent. In many cases, the ASEs that result in large MEMs also lead to large ERs. Therefore, when the ER threshold D_{er} is smaller than a certain value, even if the MEM threshold D_{mem} increases, the number of ER-MEM-feasible ASEs does not increase much and thus, the area saving of the final approximate circuits keeps almost unchanged.

When $D_{er} = 100\%$, the final approximate circuits produced by ALFANS-ER-MEM correspond to the results under the MEM constraint only. Thus, Fig. 5 also shows the performance of our algorithm when only the MEM is constrained. For CLA16 / KSA16 / RCA32 / WTM8, $D_{mem} = 300$ and $D_{mem} = 1000$ indicate that the MEM relative to the maximal output of the circuit are 0.229% / 0.229% / $3.5 \times 10^{-6}\%$ / 0.458% and 0.763% / 0.763% / $1.2 \times 10^{-5}\%$ / 1.53%, respectively. From Fig. 5, we can see that the area savings of CLA16 / KSA16 / RCA32 / WTM8 for $D_{mem} = 300$ and $D_{mem} = 1000$ are 53% / 60% / 26.6% / 37.7% and 63% / 68% / 29.2% / 53.4%, respectively.

TABLE VI
COMPARISON BETWEEN ALFANS-ER-MEM AND THE METHOD IN [19] FOR THE COMBINED CONSTRAINTS ON ERROR RATE AND MAXIMUM ERROR MAGNITUDE.

CLA16	[19]		ALFANS-ER-MEM		improvement
D_{mem}	ER(%)	area ratio(α)	ER(%)	area ratio(β)	$100\frac{\alpha-\beta}{\alpha}$(%)
300	51	0.94	50.4	0.792	15.7
300	75	0.86	73.3	0.711	17.3
300	89	0.86	86.7	0.559	35.0
300	100	0.76	99.6	0.463	39.1
1000	41	0.93	38.5	0.826	11.2
1000	70	0.87	69	0.739	15.1
1000	86	0.76	85.5	0.606	20.3
1000	100	0.655	99.9	0.363	44.6
KSA16	[19]		ALFANS-ER-MEM		improvement
D_{mem}	ER(%)	area ratio(α)	ER(%)	area ratio(β)	$100\frac{\alpha-\beta}{\alpha}$(%)
300	51	0.92	50.8	0.867	5.8
300	75	0.875	72.3	0.724	17.3
300	89	0.84	90	0.556	33.8
300	99	0.78	99.1	0.401	48.6
1000	62	0.88	58.8	0.793	9.9
1000	84	0.82	84.4	0.582	29.0
1000	91	0.76	90	0.506	33.4
1000	100	0.54	99.8	0.325	39.8
RCA32	[19]		ALFANS-ER-MEM		improvement
D_{mem}	ER(%)	area ratio(α)	ER(%)	area ratio(β)	$100\frac{\alpha-\beta}{\alpha}$(%)
300	51	0.975	49.5	0.922	5.4
300	75	0.92	75.7	0.854	7.2
300	88	0.91	88.9	0.787	13.5
1000	51	0.94	51.9	0.932	0.85
1000	76	0.89	76.3	0.864	2.9
1000	88	0.87	89	0.777	10.7
1000	100	0.75	99.9	0.703	6.3
WTM8	[19]		ALFANS-ER-MEM		improvement
D_{mem}	ER(%)	area ratio(α)	ER(%)	area ratio(β)	$100\frac{\alpha-\beta}{\alpha}$(%)
300	55	0.87	54.9	0.778	10.6
300	78	0.84	77.8	0.704	16.2
300	91	0.79	88.6	0.678	14.2
300	100	0.79	99.4	0.623	21.1
1000	9	0.94	9	0.912	3.0
1000	79	0.79	78.8	0.619	21.6
1000	89	0.76	88.2	0.599	21.2
1000	100	0.72	99.7	0.466	35.3

To further demonstrate the effectiveness of ALFANS-ER-MEM, we compared it with the method proposed in [19]. The work [19] presents the results for the same benchmarks and the same MEM constraints of $D_{mem} = 300$ and $D_{mem} = 1000$ as ours. For each benchmark and each MEM constraint, we list some results with different ERs from [19] in Table VI. We also list the results of ALFANS-ER-MEM with close ERs in the table.

In the table, the third and fifth columns show the area ratios of the method in [19] and ALFANS-ER-MEM, respectively, where area ratio is defined as the ratio of the area of the resulting approximate circuit to that of the original circuit. We can see that for all the given constraints of all the benchmarks, ALFANS-ER-MEM achieves more area saving than that of [19]. The last column shows the percentage of area improvement of ALFANS-ER-MEM over the method in [19]. On average, ALFANS-ER-MEM can further reduce the area by 19.5% over the method in [19]. We also find that the value in the last column increases almost monotonically with the ER

when the MEM constraint is fixed. This shows that ALFANS-ER-MEM achieves more improvement than the method in [19] when the ER constraint becomes more relaxed.

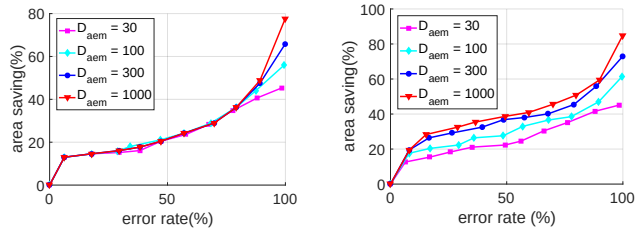
Table VII lists the average runtime of ALFANS-ER-MEM over all the tested ER thresholds for each tested MEM threshold and each benchmark. As the table shows, the average runtime for CLA16 and kSA16 is less than 1000 seconds and that for RCA32 is less than 2200 seconds. For WTM8, the average runtime varies from 2.3 hour to 6.0 hours for different MEM thresholds. In contrast, as reported in [19], the method in [19] requires more than 20 hours on a 3.4GHz workstation for RCA32 and WTM8. Thus, ALFANS-ER-MEM is more runtime efficient.

TABLE VII
AVERAGE RUNTIME IN SECONDS OF ALFANS-ER-MEM.

circuit	D_{mem}			
	30	100	300	1000
CLA16	500	605	630	590
KSA16	931	829	772	996
RCA32	2170	1365	1778	2076
WTM8	8144	11166	16209	21685

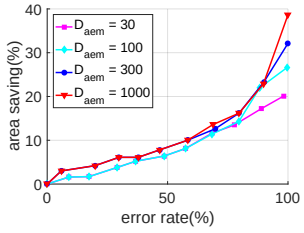
C. The Algorithm for the Combined Constraints on Error Rate and Average Error Magnitude

In this section, we study the performance of the ALS algorithm ALFANS-ER-AEM for the combined constraints on ER and AEM. We used the same set of benchmarks as those used in Section VII-B. We tested each benchmark on four different AEM thresholds D_{aem}: 30, 100, 300, and 1000. The other part of the experimental setup is same as that in Section VII-B.

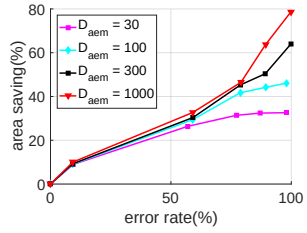


(a) CLA16

(b) KSA16



(c) RCA32



(d) WTM8

Fig. 6. Area saving versus error rate of the final approximate circuit by ALFANS-ER-AEM for different benchmarks and different average error magnitudes.

Fig. 6 plots the area saving versus ER of the final approximate circuit for each benchmark and each AEM threshold. Similar to ALFANS-ER-MEM, ALFANS-ER-AEM produces approximate circuits with smaller areas as ER and AEM thresholds increase. Comparing Fig. 5 and Fig. 6, we can see that when $D_{mem} = D_{aem}$, for the same ER threshold D_{er}, the area saving of the approximate circuit synthesized by ALFANS-ER-AEM is no less than that of the approximate circuit synthesized by ALFANS-ER-MEM. For example, as shown in Fig. 6(a), when $D_{aem} = 1000$, for $D_{er} = 0.8$, 0.9, and 1, the area saving by ALFANS-ER-AEM for CLA16 is 36.3%, 48.8%, and 77.6%, respectively. From Fig. 5(a), the area saving by ALFANS-ER-MEM for the same benchmark and D_{er}'s with $D_{mem} = 1000$ is 32.6%, 45%, and 63.7%, respectively. This is reasonable since in theory, the AEM constraint is more relaxed than the MEM constraint: a circuit satisfying an MEM threshold D must also satisfy an AEM threshold D.

Table VIII lists the average runtime of ALFANS-ER-AEM over all the tested ER thresholds for each tested AEM threshold and each benchmark. Compared to ALFANS-ER-MEM, ALFANS-ER-AEM takes more time. The reason is because in each iteration of ALFANS-ER-AEM, we build BDDs to calculate the AEMs of the circuits for each ASE that has its estimated ER within the ER constraint and has the highest score at the moment it is checked. In contrast, in each iteration of ALFANS-ER-MEM, we only build BDDs to check the ERs of the circuits in a small candidate set. Since building a BDD is a time-consuming operation, this causes a longer runtime for ALFANS-ER-AEM. From the table, we can also see that the runtime for the multiplier WTM8 is much longer than those of the adders. This is because the runtime to build a BDD for a multiplier is much longer than that for an adder.

TABLE VIII
AVERAGE RUNTIME IN SECONDS OF ALFANS-ER-AEM.

circuit	D_{aem}			
	30	100	300	1000
CLA16	597	606	659	706
KSA16	2247	2487	2634	2796
RCA32	2247	2345	2442	2623
WTM8	159272	124318	183133	96978

VIII. CONCLUSION

In this work, we presented ALFANS, a novel multi-level approximate logic synthesis (ALS) framework by approximate node simplification. The key operation of ALFANS is to remove literals in the factored-form expressions of the nodes in a Boolean network. Based on this framework, we proposed four different algorithms for three different types of error constraints. The first two algorithms, ALFANS-ER and ALFANS-ER-Fast, handle the case where only error rate is constrained. ALFANS-ER-Fast has almost the same quality as ALFANS-ER, but is more runtime-efficient. The third algorithm, ALFANS-ER-MEM, handles the combined constraints on error rate and maximum error magnitude. The fourth algorithm, ALFANS-ER-AEM, handles the combined constraints on error rate and average error magnitude. Compared to ALFANS-ER, ALFANS-ER-MEM and ALFANS-ER-AEM use more sophisticated criteria to select candidates for simplification to balance the impacts on error rate and error magnitude. Experimental results demonstrated that the ALFANS framework produces circuits with better quality than the state-of-the-art ALS approaches.

ACKNOWLEDGMENTS

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

REFERENCES

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

[2] Q. Xu, T. Mytkowicz, and N. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[3] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[4] N. Zhu, W. Goh, and K. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *International Symposium on Integrated Circuits*, 2009, pp. 69–72.

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

[6] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Design Automation Conference*, 2012, pp. 820–825.

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

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

[9] M. Shafique *et al.*, "A low latency generic accuracy configurable adder," in *Design Automation Conference*, 2015, pp. 86:1–86:6.

[10] K. Kyaw, W. Goh, and K. Yeo, "Low-power high-speed multiplier for error-tolerant application," in *International Conference of Electron Devices and Solid-State Circuits*, 2010, pp. 1–4.

[11] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *International Conference on VLSI Design*, 2011, pp. 346–351.

[12] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Design, Automation and Test in Europe*, 2014, pp. 95:1–95:4.

[13] S. Rehman *et al.*, "Architectural-space exploration of approximate multipliers," in *International Conference on Computer-Aided Design*, 2016, pp. 80:1–80:8.

[14] D. Shin and S. Gupta, "Approximate logic synthesis for error tolerant applications," in *Design, Automation and Test in Europe*, 2010, pp. 957–960.

[15] ——, "A new circuit simplification method for error tolerant applications," in *Design, Automation and Test in Europe*, 2011, pp. 1–6.

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

[17] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Automation and Test in Europe*, 2013, pp. 796–801.

[18] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *International Conference on Computer-Aided Design*, 2013, pp. 779–786.

[19] ——, "Multi-level approximate logic synthesis under general error constraints," in *International Conference on Computer-Aided Design*, 2014, pp. 504–510.

[20] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 432–444, 2014.

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

[22] A. Chandrasekharan *et al.*, "Approximation-aware rewriting of AIGs for error tolerant applications," in *International Conference on Computer-Aided Design*, 2016, pp. 83:1–83:8.

[23] Y. Wu *et al.*, "Approximate logic synthesis for FPGA by wire removal and local function change," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 163–169.

[24] Y. Yao *et al.*, "Approximate disjoint bi-decomposition and its application to approximate logic synthesis," in *International Conference on Computer Design*, 2017, pp. 517–524.

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

[26] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[27] R. Brayton *et al.*, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 6, pp. 1062–1081, 1987.

[28] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–536.

[29] S. Froehlich, D. Große, and R. Drechsler, "Approximate hardware generation using symbolic computer algebra employing Grobner basis," in *Design, Automation and Test in Europe*, 2018, pp. 889–892.

[30] R. Brayton *et al.*, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[31] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *Design, Automation and Test in Europe*, 2005, pp. 412–417.

[32] MVSIS, http://www-cad.eecs.berkeley.edu/mvsis/.

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

[34] M. Ceska *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.

[35] A. Chandrasekharan *et al.*, "Precise error determination of approximated components in sequential circuits with model checking," in *Design Automation Conference*, 2016, pp. 129:1–129:6.

[36] C. Gomes, A. Sabharwal, and B. Selman, "Model counting," in *Handbook of Satisfiability*. IOS Press, 2008, ch. 20, pp. 633–654.

[37] C. Gomes *et al.*, "From sampling to model counting," in *International Joint Conference on Artificial Intelligence*, 2007, pp. 2293–2299.

[38] P. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*. Springer, Dordrecht, 1987.

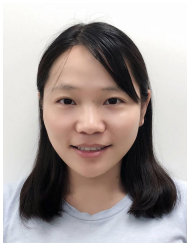
[39] E. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," University of California, Berkeley, Tech. Rep., 1992.

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

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

[42] N. Sörensson *et al.*, "Minisat v1.13 – a SAT solver with conflict-clause minimization." [Online]. Available: http://minisat.se/downloads/

[43] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0," 2015. [Online]. Available: http://vlsi.colorado.edu/ fabio/CUDD/cudd.pdf



Yi Wu is a software engineer at the verification group of Synopsys, Shanghai. She received her Ph.D. degree in Electronic and Computer Engineering at Shanghai Jiao Tong University. Her main research interests include logic synthesis/optimization for approximate computing and stochastic computing.

Weikang Qian (M'11) 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 technologies.