DALS: Delay-driven Approximate Logic Synthesis

Zhuangzhuang Zhou, Yue Yao, Shuyang Huang, Sanbao Su, Chang Meng and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
{zhouzhuangzhuang, patrickyao, King_hsy, gawaine, changmeng, qianwk}@sjtu.edu.cn

ABSTRACT

Approximate computing is an emerging paradigm for error-tolerant applications. By introducing a reasonable amount of inaccuracy, both the area and delay of a circuit can be reduced significantly. To synthesize approximate circuits automatically, many approximate logic synthesis (ALS) algorithms have been proposed. However, they mainly focus on area reduction and are not optimal in reducing the delay of the circuits. In this paper, we propose DALS, a delay-driven ALS framework. DALS works on the AND-inverter graph (AIG) representation of a circuit. It supports a wide range of approximate local changes and some commonly-used error metrics, including error rate and mean error distance. In order to select an optimal set of nodes in the AIG to apply approximate local changes, DALS establishes a *critical error network (CEN)* from the AIG and formulates a maximum flow problem on the CEN. Our experimental results on a wide range of benchmarks show that DALS produces approximate circuits with significantly reduced delays.

CCS CONCEPTS

• **Hardware → Combinational synthesis**;

KEYWORDS

approximate logic synthesis, approximate computing, delay optimization, timing optimization

1 INTRODUCTION

As modern VLSI designs encompass more complexity and transistor technology reaches nanoscale, it has been increasingly difficult to improve the performance and energy consumption of circuits by conventional design methods [20]. On the other hand, many recent applications are error tolerant by their nature. Such applications include machine learning, image processing, and multimedia. Under this circumstance, *approximate computing* was proposed as a novel circuit design paradigm [5]. Its basic idea is to modify the function of a target circuit without affecting its usability in its application. If the modification is proper, the resulting circuit will have smaller area, lower power consumption, and better performance than the original version. A topic of approximate computing related to electronic design automation is the logic synthesis for approximate circuits, which is known as *approximate logic synthesis (ALS)*. ALS seeks to synthesize an optimal approximate circuit for a target circuit with the output error under a given error constraint.

Significant progress has been made in ALS in recent years [2, 8, 9, 11, 14, 17, 18, 22, 23, 25]. A few representative works will be discussed in details in Section 2.3. However, all proposed ALS methods mainly focused on reducing the circuit area. Although the circuit delay is usually also reduced as a byproduct of these ALS methods, the potential power of ALS in improving circuit delay has not yet been fully explored. For applications such as real-time signal processing, they are error tolerant, but they also have a stringent deadline to meet. For these applications, delay, instead of area, is the primary concern. Thus, if we can develop a delay-oriented ALS flow, it will be very helpful in synthesizing circuits for these applications.

Given the difficulty in optimizing the approximate circuits globally, previous ALS methods usually repeatedly apply approximate local changes (ALCs) to the gates in a circuit until the given error constraint is reached. In these approaches, where area is the primary concern, all gates in the circuit are treated equally since they contribute to the area equally regardless of their locations in the circuit. However, this is completely different for delay minimization. Circuit delay is determined by the critical paths and only the gates on the critical paths contribute to the delay. Thus, efforts should be directed to the gates on the critical paths. Furthermore, even doing local changes repeatedly on these specific gates may not be effective. This is because there usually exist multiple critical paths of the same or nearly the same lengths. The local change may reduce the length of a particular critical path, but the lengths of the other critical paths still remain the same. Thus, to effectively reduce the delay, it should be done globally, that is, all the critical paths should be shortened simultaneously.

In this work, we propose DALS, a Delay-driven Approximate Logic Synthesis framework for delay-oriented ALS. It addresses the above-mentioned problems. First, it focuses on the gates on the critical paths. Second, it performs multiple ALCs simultaneously so that all critical paths in the circuit are shortened. DALS is a general framework that supports a wide range of ALCs. It also supports some widely-used error metrics, including error rate and mean error distance [5]. In this work, DALS is implemented on the AND-inverter graph (AIG) representation of a circuit [10]. However, it should be noted that the DALS framework can also be applied to other circuit representations. One specific challenge in the DALS framework is the selection of the sets of nodes to apply the ALCs, since there exist an exponential number of choices in a circuit. To solve this problem, we propose to establish a *critical error network (CEN)* from the AIG and then solve a maximal flow problem on the CEN. Our experimental results show that DALS produces approximate circuits with significantly reduced delays compared to the state-of-the-art ALS approaches. An application of DALS to

ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11...$15.00
https://doi.org/10.1145/3240765.3240790

synthesize approximate adders also show that DALS can produce approximate adders competitive to manual designs.

The rest of the paper is organized as follows. In Section 2, we introduce the preliminaries on AIG and error metrics and discuss some related works. In Section 3, we present the methodology of DALS. In Section 4, we present the experimental results. Finally, in Section 5, we conclude the paper.

2 PRELIMINARIES AND RELATED WORKS

In this section, we first introduce the preliminaries on AIG and error metrics and then discuss some related works.

2.1 AND-Inverter Graph

An *AND-Inverter Graph (AIG)* is a directed acyclic graph (DAG) that implements a logic function [10]. Each node in an AIG is either a primary input (PI) or a two-input AND gate. For example, Fig. 1a shows an AIG, in which the square nodes represent PIs and the round nodes represent AND gates. If a node corresponds to a two-input AND gate, we call it a *functional node*. The nodes in the AIG that give the final outputs of the logic function are also marked as primary outputs (POs) of the AIG. For example, the nodes 10 and 11 in Fig. 1a are the POs. The edges in an AIG can be complemented, indicating the inversion of the signal.

A *path* in an AIG is a sequence of connected nodes. The *length* of a path is the number of edges in the path. A *critical path* is a path with the maximum length in the AIG that starts from a PI and ends at a PO. An AIG can have multiple critical paths. The *depth* of an AIG is the length of its critical paths. For example, the depth of the AIG in Fig. 1a is 3. The *size* of an AIG is the total number of AND gates in the AIG. It is worth noting that the area and delay of the actual circuit also depends on the technology mapping of the AIG and thus, cannot be measured directly from the AIG. However, the depth and size of an AIG still correlate well with the delay and area, respectively, of the final mapped circuit.

2.2 Error Metrics

There are several common error metrics for evaluating the accuracy of approximate designs. The error metrics relevant to our work are introduced below.

Let the set of input vectors of a target circuit be $\{a_1, \dots, a_M\}$. Assume that input vector a_i ($1 \le i \le M$) occurs with a probability q_i. Let $\tilde{s_i}$ and s_i ($1 \le i \le M$) be the values encoded by the approximate and accurate output vectors, respectively, for input vector a_i.

The *error rate (ER)* of an approximate circuit is the probability for the circuit to produce an incorrect output, i.e.,

$$\text{ER} = \sum_{1 \le i \le M : \tilde{s_i} \ne s_i} q_i.$$

The *error distance (ED)* and *relative error distance (RED)* of an approximate circuit for input vector a_i ($1 \le i \le M$) are defined as

$$\text{ED}(a_i) = |\tilde{s_i} - s_i|, \quad \text{RED}(a_i) = \frac{|\tilde{s_i} - s_i|}{s_i}.$$

The *mean error distance (MED)* and *mean relative error distance (MRED)* are defined as

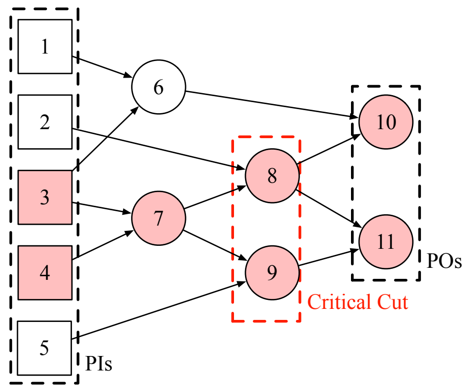
$$\text{MED} = \sum_{i=1}^{M} \text{ED}(a_i) \cdot q_i, \quad \text{MRED} = \sum_{i=1}^{M} \text{RED}(a_i) \cdot q_i.$$

2.3 Related Works

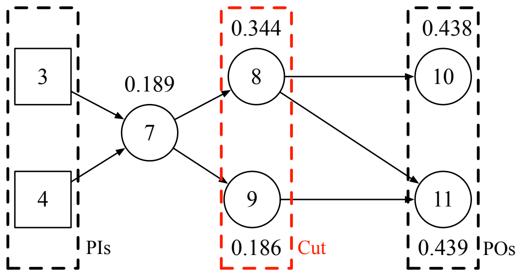
Previous works have proposed a number of ALS methods [2, 8, 9, 11, 14, 17, 18, 22, 23, 25]. By its nature, ALS is a computationally hard optimization problem, since given an error constraint, there exist numerous candidate Boolean functions satisfying the constraint, while optimizing the circuit implementation for each Boolean function by itself is a computationally hard task. To overcome this challenge, many state-of-the-art ALS techniques are based on applying ALCs to the circuits. For example, Shin and Gupta proposed to apply constant-0 and constant-1 replacement to internal gates [14]. Venkataramani *et al.* proposed to substitute one signal in the circuit by another with similar functionality [17]. Yao *et al.* proposed to perform local approximate disjoint bi-decomposition to internal signals to reduce the local area [25]. Liu and Zhang proposed some simple local circuit changes, including flipping a local output, removing a gate, and adding a gate [8] and integrated these ALCs into a stochastic optimization framework. These previous works mainly focus on area reduction, with delay reduction as a side effect. In contrast, DALS primarily focuses on delay reduction. Nevertheless, as we will show later, our DALS framework is applicable to many types of ALCs proposed in the previous works.

3 METHODOLOGY

In this section, we present the methodology of DALS. We first give an overview of DALS and then present some details of it.



(a)



(b)

Figure 1: Illustration of (a) an AIG and (b) its critical graph.

3.1 Overview

DALS works on the AIG representation of the circuit. Given an input AIG, DALS aims at synthesizing an approximate AIG with reduced depth without increasing its size, while satisfying the given error constraint. The error constraint DALS can handle includes ER constraint and MED constraint. Before we present our method, we first introduce several definitions.

The *depth* of a node in an AIG is the length of the longest path from a PI to this node. For example, the depth of node 8 in the AIG in Fig. 1a is 2. The *critical graph* $G = (V, E)$ for an AIG is a subgraph of the AIG, where V and E are the sets of nodes and edges, respectively, on the critical paths of the AIG. For example, the graph shown in Fig. 1b is the critical graph for the AIG shown in Fig. 1a. If a node in the critical graph corresponds to a PI/PO/functional node in the original AIG, we also call it a PI/PO/functional node in the critical graph. We define a *cut* for a critical graph as a set of nodes in the critical graph satisfying the following three conditions:

(1) None of the nodes in the set is a PI of the critical graph. In other words, each node in the set is a functional node.
(2) Each path from a PI node to a PO node of the critical graph passes at least one node in the set.
(3) We cannot remove any node from the set; otherwise, Condition 2 would be violated.

Note that Condition 3 essentially means that the cut is a minimal set without any redundant nodes. Based on this definition, nodes 8 and 9 form a cut for the critical graph shown in Fig. 1b. Since the critical graph is associated with an AIG, we also call a cut for a critical graph of an AIG a *critical cut* for the AIG. For example, nodes 8 and 9 form a critical cut for the AIG shown in Fig. 1a.

In order to reduce the depth of an AIG, it is not enough to shorten one critical path, because there usually exist multiple critical paths in an AIG; we need to shorten all critical paths. A critical cut for the AIG provides us a way to achieve this. Specifically, for a particular cut, if we can reduce the depth of all nodes in the cut, then the depth of the entire AIG can be reduced. For example, suppose that we are able to reduce the depths of nodes 8 and 9 in Fig. 1a both from 2 to 1, then the depth of the entire AIG reduces from 3 to 2.

Since we allow errors in approximate computing, the reduction of the depth of a node is through an ALC that could introduce error. Note that there usually exist multiple critical cuts for an AIG and multiple candidate ALCs for each node in the AIG. Different choices of the critical cut and the ALCs applied to the nodes in the selected cut will cause different error impacts on the resulting approximate AIG. In our approach, we will repeatedly select a critical cut and a set of ALCs to modify the nodes in the selected cut until the approximate AIG reaches the error constraint. Each iteration is expected to reduce the depth of the AIG by one, but at the same time, increase the error of the approximate AIG. Since we want to maximize the depth reduction, we should maximize the number of iterations. This requires that in each iteration, we should select the cut and the associated set of ALCs that lead to the minimum error impact among all choices.

The overall flow of DALS is summarized in Algorithm 1. To make the flow general, besides the input AIG and the error threshold, the flow also takes a user-specified approximation operator *op* as an input. The operator *op* should produce a set of ALCs for a node that could reduce the depth of the node. An example is replacing a node with a constant [14]. It includes two specific changes, replacement by a constant 0 and replacement by a constant 1. It can be easily seen that a constant replacement reduces the depth of the node to 0. Some other examples include the approximate substitution proposed in [17] and the approximate disjoint bi-decomposition proposed in [25], which are briefly described in Section 2.3.

By definition, a critical cut for an AIG is a cut for the critical graph of the AIG. Thus, at the beginning of each iteration, Line 4 obtains the critical graph from the current approximate AIG G_{apx}. It applies the traditional timing analysis technique to obtain the slacks of all nodes in the AIG [19] and gathers the nodes with slack 0 to form the critical graph. Then, Line 5 selects the cut for the critical graph and the associated set of ALCs that lead to the minimum error impact. The details of this step will be described in Section 3.2 below. Line 6 applies the selected set of ALCs to all nodes in the selected cut and derives the new approximate AIG G_{new}. Line 7 calculates the error between the input AIG G and G_{new}, which can be achieved through logic simulation. If the error is less than the threshold, a new round begins by assigning G_{new} to G_{apx} (see Line 3); otherwise, G_{apx} is returned.

Algorithm 1: The proposed flow of DALS.

```
Input: an AIG $G$, an error threshold $T$, and an approximation
       operator $op$.
Output: an approximate AIG $G_{apx}$ with reduced depth and error
        $e \le T$.
1 $G_{new} \Leftarrow G$;
2 while $e \le T$ do
3     $G_{apx} \Leftarrow G_{new}$;
4     $g \Leftarrow$ GetCriticalGraph($G_{apx}$);
5     $(Cut, ApxChange) \Leftarrow$ GetCutALC($g$, $op$);
6     $G_{new} \Leftarrow$ ApplyChange($G_{apx}$, $Cut$, $ApxChange$);
7     $e \Leftarrow$ GetError($G$, $G_{new}$);
8 return $G_{apx}$;
```

3.2 Selecting the Best Critical Cut and the Associated Approximate Logic Changes

A crucial step (i.e., Line 5 in Algorithm 1) in our proposed flow is to find the critical cut and the associated set of ALCs that lead to the minimum error impact. For simplicity, we call the cut *the best critical cut.* The most straightforward approach is to enumerate all critical cuts and all sets of ALCs for each cut and then choose the combination with the minimum error. However, the total number of the critical cuts grows exponentially with the size of the AIG. Furthermore, for each cut, the number of applicable ALCs grows exponentially with the size of the cut. Bare enumeration is impractical for large AIGs. To solve this issue, we propose to transform this problem into a network flow problem. This transformation relies on our proposed estimation of the error impact of applying a set of ALCs to a critical cut. Next, we will first introduce this estimation and then describe how we model the selection problem as a network flow problem.

3.2.1 Estimating the Error Impact of Applying a Set of ALCs to a Critical Cut. The most straightforward way to evaluate the error impact of a critical cut and a set of ALCs for the cut is to apply the set of ALCs to the nodes in the cut and then calculate the error of the resulting approximate AIG. However, calculating the error of an approximate AIG requires time-consuming logic simulation. If we apply this straightforward method to evaluate the error impacts of all choices of critical cuts and their ALCs, the total number of logic simulations needed is very large. To give a rough analysis on the number of logic simulations needed, we assume that the number of cuts in the critical graph is N_T, the average number of ALCs for each node is N_A, and the average size of a cut is L. Then, the number of logic simulations needed equals $N_A^L N_T$.

To reduce the complexity, we propose the following way to estimate the error impact of a set ALCs. Suppose that the critical cut contains m nodes $n_1, n_2, \ldots, n_m$ and for each $1 \leq i \leq m$, the ALC applied to node n_i is A_i. For each node n_i, we evaluate the error impact of applying ALC A_i to node n_i alone and denote the value as e_i. Then, we estimate the error impact of applying the set of ALCs to the nodes in the critical cut as the sum $e_1 + e_2 + \cdots + e_m$. This method could significantly reduce the number of logic simulations needed to evaluate the error impacts of all choices of critical cuts and their ALCs. Indeed, we only need to perform logic simulation to get the error impact for each combination of a functional node *in the critical graph* and an ALC for that node. Thus, the total number of logic simulations equals $N_A N_C$, where N_A is the average number of ALCs for a node and N_C is the number of functional nodes in the critical graph. Given that the number of cuts N_T in the critical graph is much larger than N_C, the value $N_A N_C$ is much smaller than the value $N_A^L N_T$, which is the number of logic simulations needed by the straightforward method.

Furthermore, due to the above decomposition of the total error impact into the individual error impacts, for each functional node in the critical graph, after obtaining the error impacts of all ALCs for that node, we only need to keep the ALC that gives the minimum error impact. We call this ALC the *optimal ALC* of the node and its error impact the *minimum error impact (MEI)* of the node. If a functional node is in the best critical cut, then to minimize the total error impact, we should choose to apply its optimal ALC. Thus, there is no need to consider other ALCs for the node.

After obtaining the MEI of each functional node in the critical graph, we assign that value to the node. For example, the value near each functional node shown in Fig. 1b denotes the MEI of the node. Now, the problem of selecting the best critical cut simply becomes selecting a cut for the critical graph so that the sum of the MEIs of all nodes in the cut is minimal.

The error metric for which we can apply the proposed estimation technique includes ER and MED. It should be noted that the proposed estimation may not be accurate, since the exact error impact of applying multiple ALCs together may not equal the sum of the error impacts of applying each individual ALC alone. Nevertheless, for error metrics such as ER and MED, this sum is still a good first-order approximation and enables the design of a more efficient algorithm.

3.2.2 *Selecting the Best Critical Cut.*

In this section, we present a method to select the best critical cut. By our estimation method for the error impact, the best critical cut is a cut for the critical graph so that the sum of the MEIs of all nodes in the cut is minimal. Our method first maps the original critical graph into a *critical error network (CEN)* and then solves a network flow problem on the CEN.

The CEN is built from the critical graph. We also need to assign proper capacities to the edges in CEN. The details for building the CEN is shown below.

1. For each functional node n with MEI e in the critical graph, we add a pair of nodes n_a and n_b to the CEN. We also add an edge from n_a to n_b with capacity of e to the CEN.
2. For each edge from a functional node u to a functional node v in the critical graph, we add an edge from u_b to v_a with infinite capacity to the CEN.
3. We add a source node s. For each edge from a PI node p to a functional node n in the critical graph, we add an edge from s to n_a with infinite capacity to the CEN.
4. We add a sink node t. For each PO node q in the critical graph, if it is a functional node, then we add an edge from q_b to t with infinite capacity to the CEN.

The CEN built from the critical graph shown in Fig. 1b is given in Fig. 2.

The CEN is a classic flow network [3]. For a flow network, a *cut* is defined as a set of edges that disconnects the source and sink upon removal. The *capacity* of a cut is the total capacity of all edges in the cut. A *minimum cut* of a flow network is a cut with the minimum capacity over all cuts of the flow network. Given the above mapping procedure, it is easily seen that the problem of selecting the cut for the critical graph with the minimum sum of MEIs now reduces to the problem of finding a minimum cut in the CEN. By the max-flow min-cut theorem [3], the capacity of a minimum cut in a flow network equals the maximum flow of the network. Thus, we can find a minimum cut of the CEN by solving the maximum flow problem on the CEN.

Once we have identified each edge in the minimum cut, we can get the corresponding nodes in the critical graph from the mapping relation and obtain the cut in the critical graph that gives the minimum sum of MEIs.

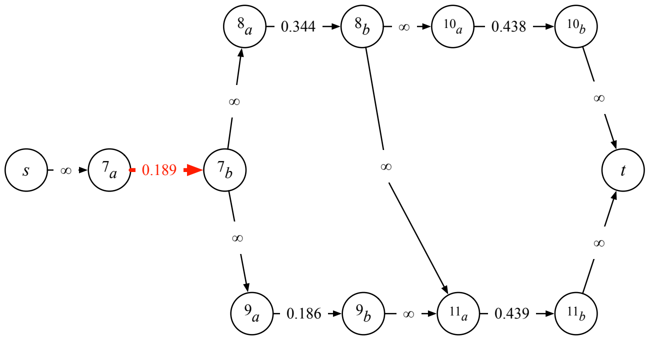


Figure 2: The critical error network built from the critical graph shown in Fig. 1b.

3.2.3 *The Entire Flow.*

Algorithm 2 shows the entire flow for finding the best critical cut and the associated set of ALCs, namely the function *GetCutALC* in Algorithm 1. Lines 1–5 iterate over each node in the critical graph and find the optimal ALC and the MEI of each node. Line 6 builds the CEN from the critical graph, given the

MEI of each node in the critical graph. Line 7 solves the maximal flow problem and returns the minimum cut in the CEN. In our implementation, we use Dinic's algorithm to solve the maximal flow problem [4]. Line 8 maps the minimum cut in the CEN into the cut *Cut* in the critical graph. Line 9 obtains the set of optimal ALCs for all nodes in *Cut*.

Algorithm 2: The flow of the function *GetCutALC* for finding the best critical cut and the associated set of ALCs.

```
Input: a critical graph g and an approximation operator op.
Output: the critical cut Cut and the associated set of ALCs
        ApxChange that lead to the minimum error impact.
1  foreach node n in graph g do
2      foreach ALC x of node n generated by the approximation operator op do
3          obtain the error impact of applying ALC x to node n;
4      n.e ⇐ the minimum error impact over all ALCs of n;
5      n.OptmALC ⇐ the ALC of n with the minimum error impact;
6  f ⇐ BuildCEN(g);
7  minCut ⇐ SolveMaxFlow(f);
8  Cut ⇐ EdgetoNode(minCut);
9  ApxChange ⇐ GetBestALC(Cut);
10 return Cut and ApxChange;
```

4 EXPERIMENTAL RESULTS

In this section, we present the experimental results of our proposed DALS method. We implemented the algorithm in C++. All experiments were conducted on a laptop computer with Intel i7-5700HQ CPU @ 2.70GHz and 16 GB memory running Linux 4.13.

In our implementation, the errors were measured by performing logic simulations. We assumed that all primary input vectors have equal probabilities. For each logic simulation, we generated 100000 input vectors randomly, which is sufficient to get the target error metrics such as ER and MED with high accuracy. The proposed DALS algorithm takes a user-specified approximation operator. We set it as the constant replacement operator in our experiments, which includes two ALCs, namely constant-0 replacement and constant-1 replacement. The areas and delays of the circuits were reported after performing technology mapping by the logic synthesis tool ABC [1] using the MCNC generic standard cell library [24]. The reported areas are normalized to that of a unit-strength inverter. The unit of the reported delays is *ns*.

We performed two sets of experiments to evaluate DALS. In the first set, we applied DALS to some common benchmarks with the error metric as ER. In the second set, we applied it to synthesize approximate adders with the error metric as MED.

4.1 Study on Common Benchmarks under Error Rate Constraint

In this set of experiments, we selected several ISCAS85 benchmarks and an arithmetic circuit ALU4 synthesized by Synopsys Design Compiler [16]. The benchmark information is listed in Table 1. We chose ER as the error metric.

For each benchmark, we first applied DALS to get the corresponding approximate AIG with reduced depth. DALS works iteratively.

Table 1: Benchmark circuit information.

circuit	#I/Os	#nodes	area	delay
C880	60/26	238	609	16.50
C1355	41/32	211	781	16.20
C1908	33/25	290	831	23.20
C3540	50/22	729	1838	26.40
C5315	178/123	1026	2557	24.10
C7552	207/107	1466	3930	25.20
ALU4	14/8	1120	2776	12.80

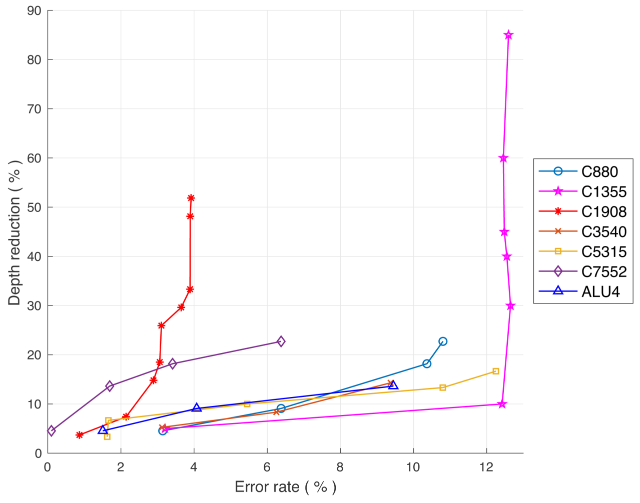


Figure 3: Depth reduction rate versus error rate by DALS.

We recorded the result after each iteration. The experimental results are shown in Fig. 3, which illustrates the relationship between depth reduction rate and ER for all the benchmarks. We can see that DALS can reduce the depth of the AIG when some inaccuracy is allowed and the depth reduction rate increases with ER.

We further took one point on the depth-reduction-rate-versus-ER curve for each benchmark and performed technology mapping to the approximate AIG to obtain the area and delay of the final mapped circuit. Table 2 lists the actual ERs, area reduction rates, and delay reduction rates of the approximate circuits in the 2nd, 3rd, and 4th columns, respectively. We can see that for all benchmarks, DALS can reduce the delay of the final mapped circuit when some inaccuracy is allowed. For benchmarks C880, C1355, and C1908, the circuit delays can be reduced dramatically compared to the introduced ER. For some other benchmarks, the delay reduction is not so significant compared to the introduced ER. We believe this is due to the usage of the constant replacement as the approximation operator in our experiment. It reduces the AIG depth by one in each round, but in some cases, the introduced ER could be pretty large. We believe that if some other approximation operators that can produce ALCs with smaller introduced ER are applied, we can have better experimental results. We will explore the effects of some other approximate operators in our future work. The possible candidates are approximate substitution [17] and the approximate disjoint bi-decomposition [25]. They could produce ALCs that have smaller ERs and at the same time, reduce the node depth.

DALS focuses on the delay reduction, and the area reduction is just a side effect of it. We only expected the area of the approximate circuit to be no more than that of the original one. However, as Table 2 shows, the area of the synthesized approximate circuit also decreases significantly for some benchmarks.

Table 2: DALS results and comparison with a state-of-the-art area-driven ALS method [15].

		DALS		[15][‡]	
circuit	error rate	Δarea[†]	Δdelay[†]	Δarea	Δdelay
C880	10.73%	17.90%	33.33%	24.04%	16.67%
C1355	12.48%	95.83%	93.83%	41.68%	2.53%
C1908	3.78%	58.24%	55.60%	58.63%	45.14%
C3540	14.31%	19.80%	16.37%	35.67%	8.33%
C5315	15.98%	3.01%	19.92%	13.10%	0.90%
C7552	6.38%	4.43%	16.90%	21.79%	0.91%
ALU4	9.45%	33.86%	19.23%	68.67%	7.69%

[†] ΔArea and ΔDelay are the area reduction rate and delay reduction rate, respectively, of the approximate circuit with respect to the original circuit.

[‡] The results are post-processed by delay-driven traditional logic synthesis.

To study the effectiveness of DALS, we also compared it with a state-of-the-art area-driven ALS approach [15]. Previous area-driven ALS approaches have been proved to be effective in reducing circuit area. However, they do not focus on the delay. To make it fair, after a circuit had been synthesized by the ALS method in [15], we further applied delay-driven traditional logic synthesis from ABC to minimize the delay of the approximate circuit. To reach the limit, we set the delay constraint as low as possible. For a fair comparison, we synthesized each benchmark by the method in [15] so that the actual ER of the approximate circuit is very close to the ER reported in column two of Table 2. The results of the comparison study are shown in the last two columns in Table 2.

From the results, we can see that even after subsequent delay-driven traditional logic synthesis, the state-of-the-art area-driven ALS method is not optimal in reducing delay. In certain cases, although the areas of the circuits reduce significantly, the delays still stay nearly the same. This is reasonable because an area-driven ALS method does not specifically optimize the nodes on the critical paths and it may choose other nodes to perform approximate changes. In contrast, DALS reduces more delay than the area-driven ALS method. Thus, DALS can provide a better solution when delay is the primary goal. Since area reduction is just a side effect of DALS, it may not be as much as that of the area-driven ALS method. However, for benchmark C1355, DALS can even reduce more area.

4.2 Study on Approximate Adders under Mean Error Distance Constraint

In this set of experiments, we applied DALS to two accurate adder designs RCA_N8 and RCA_N16, which are 8-bit and 16-bit ripple carry adders, respectively, to generate the corresponding 8-bit and 16-bit approximate adders. We used Yosys open synthesis suit [21] to convert the Verilog HDL codes of adders to BLIF files as the inputs to our program. We used MED, which is one of the most widely-used error metrics for approximate adders [6], as the error metric in DALS.

The experimental results are shown in Table 3. The top and bottom halves of the table show the results for the 8-bit and the 16-bit adders, respectively. The synthesized adders were compared to two types of previously proposed manually designed approximate adders. They are *generic accuracy configurable adder (GeAr)* [12] and *accuracy-configurable adder (ACA)* [7]. In the table, the adders with the names starting with "GeAr" are GeArs, while the one with the name starting with "ACA" is an ACA. These designs were taken from an online repository [13] and the meaning of the abbreviations is given in that repository. We chose these specific designs for comparison, since they locate on a Pareto optimal quality-versus-error curve according to the results reported in [2]. We obtained their areas and delays by the logic synthesis tool ABC using the same setup as the experiments for DALS. The adders synthesized by DALS are those with the names beginning with "DALS". Since there are three manually designed approximate adders for both the 8-bit adder and the 16-bit adder categories, we also show three approximate adders synthesized by DALS with different trade-offs between delay and MED. To give a comprehensive accuracy evaluation, besides MED, we also list the ER and MRED for each approximate adder.

Table 3: Approximate adders synthesized by DALS and comparison with some manually designed approximate adders.

circuit	ER	MRED	MED	area	delay
RCA_N8	0.00%	0.0000%	0.00	140	10.2
GeAr_N8_R2_P4	2.37%	0.6924%	1.50	138	8.6
DALS_N8_1	22.64%	0.5638%	1.07	134	8.4
GeAr_N8_R2_P2	18.73%	3.674%	7.52	128	7.0
DALS_N8_2	39.47%	2.804%	5.45	131	6.6
GeAr_N8_R1_P2	30.05%	7.104%	15.29	124	5.4
DALS_N8_3	69.92%	6.067%	13.61	85	5.4
RCA_N16	0.00%	0.0000%	0.00	315	13.4
GeAr_N16_R4_P4	5.86%	0.2657%	124.4	299	10.2
DALS_N16_1	51.85%	0.2321%	115.9	280	10.0
GeAr_N16_R2_P4	11.65%	0.9819%	510.4	290	8.6
DALS_N16_2	67.31%	1.0752%	514.6	269	8.2
ACA_II_N16_Q4	48.16%	3.893%	2049	260	7.0
DALS_N16_3	87.70%	3.024%	2043	207	7.0

The experimental results show that DALS can generate highly competitive approximate adders with reduced delays. In most cases, the adders synthesized by DALS have better areas, delays, MEDs, and MREDs than the previous manual designs at the cost of higher ERs. Thus, DALS can provide a better solution for many real-world applications where MED and MRED are more important than ER, such as image processing and machine learning. The fact that DALS generates adders with smaller MEDs but larger ERs is because the error metric is set as MED in DALS. Given this error metric, DALS tends to approximate the logic that affects the less significant outputs of the adder and it ignores the influence to ER.

Finally, we compared DALS to a previous ALS flow [2] that was also tested on approximate adders. The ALS flow [2] is based on approximation-aware rewriting of AIGs. We compared the quality of the 16-bit approximate adders synthesized by DALS to that by the previous flow [2]. The comparison results are shown in Table 4, where the results for the work [2] were copied from [2]. To make it fair, we used the same 16-bit ripple carry adder design and the same setup as the experiments in [2] when testing DALS.

Table 4: Comparison between DALS and the work in [2] on the synthesis of 16-bit approximate adders.

circuit	ER/%	WCE	MBF	area	delay	area×delay	runtime/s
appx9	99.80	2038	9	254	13.4	3403.6	229
appx11	96.88	496	5	277	13.4	3711.8	201
appx13	99.22	1024	7	264	13.4	3537.6	220
DALS_N16_1	51.85	340	7	280	10.0	2800.0	6
appx12	99.90	4090	11	226	12.7	2870.2	187
DALS_N16_2	67.31	5380	11	269	8.2	2205.8	16
appx8	99.64	8320	13	120	7.0	840.0	151
appx10	99.64	8320	13	120	7.0	840.0	150
DALS_N16_3	87.70	6144	9	207	7.0	1449.0	23

Instead of MED, the previous work [2] chose to use the worst-case error (WCE) and maximum bit-flip (MBF) error to evaluate the quality of the approximate adders. The WCE is defined as the maximum error distance over all input vectors, while the MBF is defined as the maximum hamming distance between the approximate output vector and the accurate output vector over all input vectors. For comparison purpose, we also list these two metrics in the 3rd and 4th columns in Table 4. In the table, the adders with the names starting with “appx” are those 16-bit approximate adders taken from [2]. We arrange them into three groups by their delays. Based on WCE and MBF, it can be seen that the accuracy of the adders decreases from group 1 to group 3. In each group, we also show one approximate adder synthesized by DALS, which has its name starting with “DALS”.

From the table, we can see that in both group one and group two, the adder synthesized by DALS outperforms the corresponding approximated adder(s) from [2] in at least two error metrics among the three, i.e., ER, WCE, and MBF. The DALS adder has smaller delay but larger area than the adder(s) from [2]. However, in terms of area-delay product (ADP), which is a more comprehensive measure on the hardware quality, the DALS adder is much better. In group one, the DALS adder reduces the ADP by at least 17.7%, while in group two, the DALS adder reduces the ADP by 23.1%. In group three, where the accuracy of adders is much lower than that of the adders in the previous two groups, the DALS adder outperforms the approximated adders from [2] in all three error metrics but at a higher cost of area and ADP. In conclusion, the approximate adders synthesized by DALS are better than those synthesized by the ALS method in [2] when the accuracy requirement is high. The last column in the table lists the runtime. It should be noted that our testing platform is slower than that used in [2]. Thus, we can conclude that DALS is much faster than the previous method.

5 CONCLUSION

In this work, we proposed DALS, a delay-driven approximate logic synthesis framework, which can produce approximate circuits with significantly reduced delays. It supports a wide range of approximate local changes and some commonly-used error metrics, including error rate and mean error distance. Its basic idea is to establish a critical error network (CEN) for the AIG representation of a target circuit and utilize the CEN to select the optimal set of nodes to apply depth-reduction approximate local changes. We tested DALS on a wide range of benchmarks with different error metrics. Although we only used the trivial constant replacement as the approximate local changes, the experimental results show that DALS outperforms the state-of-the-art area-driven ALS approaches, even after subsequent delay-driven traditional logic synthesis. Thus, DALS is a promising solution for delay-oriented tasks. In our future work, we will apply more sophisticated approximation local changes to DALS to enhance its performance.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. 2018. ABC: A system for sequential synthesis and verification, release 70330. (2018). http://www.eecs.berkeley.edu/~alanmi/abc
[2] A. Chandrasekharan, M. Soeken, et al. 2016. Approximation-aware rewriting of AIGs for error tolerant applications. In *ICCAD*. 83:1–83:8.
[3] T.H. Cormen, C.E. Leiserson, et al. 2001. *Introduction to algorithms.* MIT Press.
[4] E.A. Dinic. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math* 11 (1970), 1277–1280.
[5] J. Han and M. Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS*. 1–6.
[6] H. Jiang, J. Han, and F. Lombardi. 2015. A comparative review and evaluation of approximate adders. In *GLSVLSI*. 343–348.
[7] A.B. Kahng and S. Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *DAC*. 820–825.
[8] G. Liu and Z. Zhang. 2017. Statistically certified approximate logic synthesis. In *ICCAD*. 344–351.
[9] J. Miao, A. Gerstlauer, and M. Orshansky. 2014. Multi-Level approximate logic synthesis under general error constraints. In *ICCAD*. 504–510.
[10] A. Mishchenko, S. Chatterjee, and R. Brayton. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC*. 532–535.
[11] A. Ranjan, A. Raha, et al. 2014. ASLAN: Synthesis of approximate sequential circuits. In *DATE*. 364:1–364:6.
[12] M. Shafique, W. Ahmad, et al. 2015. A low latency generic accuracy configurable adder. In *DAC*. 86:1–86:6.
[13] M. Shafique, W. Ahmad, et al. 2018. Library of Approximate Adders. http://ces.itec.kit.edu/GeAR.php. (2018).
[14] D. Shin and S. K. Gupta. 2011. A new circuit simplification method for error tolerant applications. In *DATE*. 1–6.
[15] S. Su, Y. Wu, and W. Qian. 2018. Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis. In *DAC*. 54:1–54:6.
[16] Synopsys Inc. 2018. http://www.synopsys.com. (2018).
[17] S. Venkataramani, K. Roy, and A. Raghunathan. 2013. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *DATE*. 1367–1372.
[18] S. Venkataramani, A. Sabne, et al. 2012. SALSA: Systematic logic synthesis of approximate circuits. In *DAC*. 796–801.
[19] J. Vygen. 2006. Slack in static timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 9 (2006), 1876–1885.
[20] M. M. Waldrop. 2016. The chips are down for Moore’s law. *Nature* 530, 7589 (2016), 144–147.
[21] Clifford Wolf. 2016. Yosys open synthesis suite. (2016).
[22] Y. Wu and W. Qian. 2016. An efficient method for multi-level approximate logic synthesis under error rate constraint. In *DAC*. 128:1–128:6.
[23] Y. Wu, C. Shen, et al. 2017. Approximate logic synthesis for FPGA by wire removal and local function change. In *ASPDAC*. 163–169.
[24] S. Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center of North Carolina (MCNC).
[25] Y. Yao, S. Huang, et al. 2017. Approximate disjoint bi-decomposition and its application to approximate logic synthesis. In *ICCD*. 517–524.