DASALS: Differentiable Architecture Search-Driven Approximate Logic Synthesis

Xuan Wang[1], Zheyu Yan[2], Chang Meng[1], Yiyu Shi[2], and Weikang Qian[1,3]
[1]UM-SJTU Joint Inst. and [3]MoE Key Lab of AI, Shanghai Jiao Tong University, China; [2]University of Notre Dame, USA
Emails: xuan.wang@sjtu.edu.cn, zyan2@nd.edu.cn, changmeng@sjtu.edu.cn, yshi4@nd.edu, qianwk@sjtu.edu.cn

***Abstract*—Approximate computing is a promising computing paradigm for designing energy-efficient systems. To automatically generate approximate circuits, many local iterative approximate logic synthesis (ALS) methods have been proposed. They need to specify a particular local approximation change and apply it to modify the local structure of a circuit in each round. This will lose some global optimization opportunities, thus, degrading circuit quality. In this paper, we propose DASALS, a differentiable architecture search-driven ALS method, to directly search the whole circuit structure to obtain the approximate circuits with better circuit quality-accuracy trade-off. DASALS is based on a proper continuous relaxation of the discrete search space of ALS and an efficient gradient descent-based search algorithm. The experimental results show that compared with a state-of-the-art method, DASALS on average reduces the area-delay product by 10.82% and mean square error by 10.93%.**

***Index Terms*—approximate computing, approximate logic synthesis, differentiable architecture search**

I. Introduction

In the era of nano-scale transistors [1], it is crucial to reduce power consumption for modern computing systems. Meanwhile, many applications widely used today exhibit error resilience, such as image processing and machine learning. This trend leads to a promising paradigm for designing energy-efficient computing systems, known as *approximate computing* [2]–[4]. Its basic idea is to introduce a small amount of error to modify the function of the system, which leads to improvement in circuit area, delay, and power consumption.

A key research area of approximate computing is *approximate logic synthesis (ALS)*. Given an original circuit and an error constraint, ALS automatically synthesizes an approximate circuit with a small hardware cost satisfying the error constraint. Existing ALS methods can be categorized into local methods and global methods.

A local ALS method modifies local structures in a circuit. Such a modification is called a *local approximate change (LAC)*. Most local methods are iterative, which obtains the final approximate circuit by multiple rounds of LACs [5]–[11]. For instance, the work [5] proposes an ALS approach named SASIMI to synthesize approximate circuits. Its LAC is to replace a signal with another signal or its negation in the circuit. Hashemi *et al.* proposed BLASYS, which partitions a circuit into subcircuits and approximates each subcircuit by Boolean matrix factorization [9]. The works [8], [11] propose efficient approaches to speed up the local iterative ALS methods.

A global ALS method directly searches the whole circuit structure to obtain an approximate circuit [12]–[14]. For instance, the work [12] proposes a method based on Cartesian genetic programming to generate approximate circuits. In [13], an exact synthesis-based method called MinAC is proposed to automatically generate the whole approximate circuits with minimal area.

Compared to local ALS methods, global ones do not require specifying a particular LAC, thus offering more optimization opportunities

This work is supported in part by the National Key R&D Program of China under Grants 2021ZD0114701 and 2020YFB2205501, the National Natural Science Foundation of China under Grants T2293700 and T2293701, and ACCESS–AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR. Corresponding authors: Yiyu Shi and Weikang Qian.

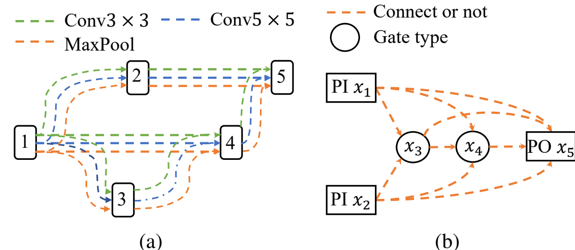

Fig. 1. Search space of (a) NAS for CNN and (b) ALS.

and exploring a larger design space. Ideally, global ALS methods would result in a better circuit quality. However, existing global ALS methods either have a scalability issue [12], [13] or do not achieve a good circuit quality [14].

To address the above issues of global ALS methods, we adopt the idea of *neural architecture search (NAS)* to synthesize approximate circuits in this work. NAS is an automated technique for discovering the optimized neural network (NN) architectures for a given task, *e.g.*, maximizing the accuracy on CIFAR-10 dataset while minimizing the latency [15]. Fig. 1(a) illustrates the search space of NAS for a *convolutional NN (CNN)*. It is a *directed acyclic graph (DAG)*, where each node corresponds to a feature map, and each directed edge from node i to j is associated with some operations that transform i to j, such as convolution and pooling operations. The goal of NAS is to determine the most suitable operation for each edge. The main methods for NAS include reinforcement learning [16], [17], evolutionary algorithms [18], and gradient-based methods [15], [19]. Gradient-based methods outperform the others in terms of faster convergence [19]. They are based on a continuous relaxation of the discrete architecture representation, making the search space differentiable. This allows for an efficient optimization of the architecture using gradient descent techniques [19]. NAS and ALS share some similarities in that they both involve searching a large design space for optimal architectures while balancing trade-offs among different performance metrics. However, they also have notable differences. Fig. 1(b) shows the search space of ALS, which is also a DAG. The squares represent *primary inputs (PIs)* and *primary outputs (POs)*, while the circles denote gates. The aim of ALS is to determine the actual interconnections among nodes and the type of each gate (*e.g.*, AND or OR gates), to minimize circuit error and hardware cost simultaneously. Thus, ALS has a more complex search space than NAS. It is impractical to directly apply the gradient-based NAS methods to solve the ALS problem due to the following challenges.

- NAS only needs to determine the operation for each edge in its search space to obtain an NN architecture. However, ALS needs to determine the node connections and gate type for each gate in a circuit. Thus, the search space of ALS is much larger.
- For NAS, each layer of an NN performs *continuous* arithmetic operations. Once the discrete architecture representation is relaxed into a continuous form, we can efficiently search the architecture by gradient descent. However, for ALS, each gate performs logic operation, which is discrete. We not only need to make the *discrete*

circuit choices (*i.e.*, node connections and the type of each gate) continuous, but also the logic operations of gates continuous.

- For the complex search space of ALS, we need to design a new efficient search algorithm.

To address the above challenges, in this work, we propose DASALS, a differentiable architecture search-driven ALS method, to directly search the whole circuit structure to generate the approximate circuits with better circuit quality-accuracy trade-off. The main contributions are as follows.

1) To model the discrete search space of ALS, we propose a serial circuit structure representation for small circuits. In addition, we propose a parallel circuit structure representation with implicit skip connections for large circuits to make search efficient.
2) To make the discrete circuit choices (*i.e.*, node connections and the type of each gate) of ALS and logic operations of gates continuous, we propose a continuous relaxation scheme.
3) To guide the search towards architectures with better circuit quality-accuracy trade-off, we define a loss function considering both the circuit error and hardware cost. In addition, we propose a differentiable method to roughly estimate the circuit area.
4) To minimize the loss function, we propose a novel search algorithm based on gradient descent.
5) To make the search algorithm computationally feasible and efficient, we propose some implementation tricks, including initializing from a good baseline, escaping from the bad local minimum, and avoiding redundant computation.

DASALS can be applied to any input distribution and any *differentiable* error and hardware cost metrics. As an example, this work considers *mean square error (MSE)* as the error metric. MSE measures the average of squared differences between the approximate and accurate outputs. The experimental results show that compared with a state-of-the-art ALS method [8], DASALS on average reduces the *area-delay product (ADP)* by 10.82% and MSE by 10.93%. The rest of the paper is organized as follows. Section II presents the DASALS methodology. Section III shows the experimental results. Section IV concludes the paper.

II. Methodology

This section presents the methodology of DASALS. First, we present a model of the discrete search space of ALS in Section II-A. To apply the gradient descent methods for efficient architecture search, we then propose a continuous relaxation scheme for the discrete search space of ALS (Section II-B). To guide the search towards architectures that balance error with hardware cost, we define a loss function in Section II-C. Subsequently, we design a novel search algorithm based on gradient descent to minimize the loss function to obtain the feasible approximate circuits (Section II-D. Finally, we propose some implementation tricks to make the search algorithm computationally feasible and efficient (Section II-E).

A. Search Space Modeling

In this work, we use an OR-inverter graph (OIG) to represent a circuit, in order to minimize the number of gate types we can choose from to reduce the search space. Note that this is just an intermediate representation, and the final circuit is mapped from its OIG based on a gate library. OIG is a DAG consisting of PI nodes, two-input nodes representing logical OR operation, and edges with optional markers indicating logical negation. In what follows, we call a two-input node in an OIG *an OR node*, *an OR gate*, or *a gate* for short. Note that we do not consider the more widely-used AND-inverter graph [20]. This is because when applying the gradient descent method for architecture search, an AND operation is treated as a multiplication operation. If any input to an AND gate is zero, the output is zero, leading to zero gradient during backpropagation, which can cause the vanishing gradient problem [21].

Assume that an accurate circuit has n PIs and m POs. In our method, given an integer r, we want to find an OIG of r gates with a small error and hardware cost. Note that the *actual* number of OR gates in the obtained OIG can be even less than r. This is because some OR nodes are degraded to buffers or inverters depending on their input connections, as we will describe later. To obtain an OIG, we need to determine the node connections, whether to negate each edge in the OIG, and which node each PO binds to. As the OIG has n PIs and r gates, we let $x_1, \dots, x_n$ be the PIs and $x_{n+1}, \dots, x_{n+r}$ be the gates. Denote the two inputs of gate x_i as y_{ij}, where $j = 1, 2$. Denote the output of PI/gate x_i as y_i. Next, we design two types of OIG architectures: serial and parallel OIG architectures.

1) Serial OIG Architecture: To obtain a serial OIG architecture, we need the PI number n, PO number m, and a reference approximate circuit, which serves as a structural template upon which the serial OIG architecture is built. The reference approximate circuit can be obtained from an existing ALS method, such as [8], [9]. First, we convert the reference circuit into a *reference serial OIG architecture* based on its topological sorting order. We place n PIs at the PI layer and then let each subsequent layer contain only a single gate in a topological order. To reduce the search space, we assume that all the m POs bind to the last m nodes in the descending order of significance. If a PO binds to node x_i which is not one of the last m nodes, we add a buffer at the end of the circuit with x_i as its input, and bind the PO to this buffer. Then, we set the gate number r as the total gate number of the reference serial OIG architecture. Given the reference serial OIG architecture, we build the serial OIG architecture. First, we renumber the nodes after performing a topological sort. Then, we establish the potential connections, where the *candidate fanin nodes* of each gate x_i include all the former nodes x_j $(1 \le j < i)$.

Example 1. *Fig. 2(a) illustrates a serial OIG architecture with* 2 *PIs,* 1 *PO, and* 3 *gates. Nodes* x_1 *and* x_2 *are the PIs placed at the PI layer. Nodes* x_3*,* x_4*, and* x_5 *are the gates, which are placed at layers* 1*,* 2*, and* 3*, respectively. In addition, node* x_5 *is the PO. The blue, orange, and green dashed lines represent possible input connections for gates at layers* 1*,* 2*, and* 3*, respectively. For example, the candidate fanin nodes of gate* x_5 *at layer* 3 *are* x_1, x_2, x_3*, and* x_4*.*

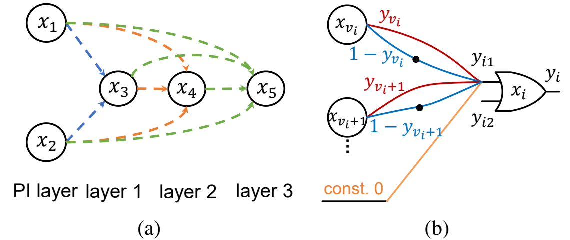

Fig. 2. (a) A serial OIG architecture; (b) three types of candidate input signals of the input y_{i1} of gate x_i.

Next, we show how to obtain the candidate signals for an input of gate x_i. Denote the total number of candidate fanin nodes of gate x_i as K_i, and among them, the one with the smallest ID is x_{v_i}. For gate x_i in a serial OIG architecture, $K_i = i - 1$ and $x_{v_i} = x_1$. As shown in Fig. 2(b), each input y_{ij} $(1 \le j \le 2)$ of gate x_i can connect with three types of candidate signals: 1) the output y_k of any node x_k, where $v_i \le k < v_i + K_i$ (denoted as the red lines in Fig. 2(b)); 2) the negation of y_k, *i.e.*, $1 - y_k$, where $v_i \le k < v_i + K_i$ (denoted as the blue lines with black markers in Fig. 2(b), where the black markers

indicate logical negation); 3) constant 0 (denoted as the orange line in Fig. 2(b)), which is used to enrich the architecture representation. Note that the other constant, constant 1, is already included in the OIG, which is realized by the OR of y_k and $1 - y_k$. Thus, the number of candidate signals for an input of gate x_i is $2K_i + 1$.

Example 2. *For the serial OIG architecture shown in Fig. 2(a), we obtain the candidate signals for an input of gate x_4. The candidate fanin nodes of x_4 are x_1, x_2, and x_3. Thus, $K_4 = 3$. Hence, the number of candidate signals for an input of gate x_4 is $2K_4 + 1 = 7$. They are x_1, x_2, x_3, $1 - x_1$, $1 - x_2$, $1 - x_3$, and constant 0.*

After obtaining the candidate signals for the input y_{ij} of gate x_i, the input y_{ij} is formulated as:

$$y_{ij} = \sum_{k=1}^{2K_i+1} w_{ijk} z_k, \tag{1}$$

$$z_k = \begin{cases} y_{k+v_i-1}, 1 \leq k \leq K_i \\ 1 - y_{k-K_i+v_i-1}, K_i + 1 \leq k \leq 2K_i \\ 0, k = 2K_i + 1 \end{cases} \tag{2}$$

$$\text{s.t.} \sum_{k=1}^{2K_i+1} w_{ijk} = 1; w_{ijk} \in \{0, 1\}, \tag{3}$$

where w_{ijk} is a *connection indicator variable*, indicating whether y_{ij} connects with the signal z_k. If it does, $w_{ijk} = 1$; otherwise, $w_{ijk} = 0$. Eq. (3) ensures that each input of gate x_i only connects with one candidate signal, as required in an actual circuit. For $1 \leq k \leq 2K_i + 1$, variables w_{ijk}'s form a *connection indicator vector* $\mathbf{w}_{ij}$. For $n+1 \leq i \leq n+r$ and $1 \leq j \leq 2$, these vectors $\mathbf{w}_{ij}$'s form a three-dimensional *connection indicator tensor* $\mathbf{W}$, which represents the whole architecture of the OIG circuit within the discrete space.

Given the inputs y_{ij} $(j = 1, 2)$ of gate x_i, its output y_i is calculated based on logical OR:

$$y_i = y_{i1} \vee y_{i2}. \tag{4}$$

The serial OIG architecture search problem is to find a suitable set of connection indicator variables w_{ijk} to generate a good approximate circuit. To measure the complexity of this problem, we compute the total number of connection indicator variables w_{ijk}. For each input y_{ij} of gate x_i, the number of indicator variables is $2K_i + 1$. Since each gate has 2 inputs, the number of indicator variables for gate x_i is $2 \times (2K_i + 1)$. For all the gates x_i $(n + 1 \leq i \leq n + r)$ in the OIG, the number of indicator variables is:

$$\sum_{i=n+1}^{n+r} 2 \times (2K_i + 1). \tag{5}$$

For the serial OIG architecture, $K_i = i - 1$. Thus, its number of connection indicator variables is $\sum_{i=n+1}^{n+r} 2 \times (2 \times (i - 1) + 1) = 2r^2 + 4nr$. Thus, the total number of connection indicator variables of the serial architecture grows quadratically with the gate number r, making the search of serial architecture inefficient for a large OIG.

2) Parallel OIG Architecture: To efficiently search the OIG architecture for large designs, we propose a parallel OIG architecture, where only nodes of adjacent layers are connected. It greatly reduces the number of connection indicator variables in the search space.

To build a parallel OIG architecture, we need the PI number n, PO number m, and a reference approximate circuit, which serves as a structural template upon which the parallel OIG architecture is built. Similar to the serial OIG architecture, to reduce the search space, we assume that all the POs bind to the last m nodes.

To build the parallel OIG architecture, we first convert the reference circuit into a *reference parallel OIG architecture* satisfying that its POs are placed at the last layer, denoted as layer l. To achieve this, if a PO binds to node x_i at layer p $(p < l)$ originally, we add a chain of $(l - p)$ buffers after node x_i and binds the PO to the last buffer in the chain so that the PO binds to a node at layer l. Fig. 3(a) illustrates a reference parallel OIG architecture with $n = 2$ and $m = 1$, where x_1 and x_2 are the PIs, and x_5 at the last layer is the PO. However, there usually exist cross-layer connections in a reference parallel OIG architecture. For example, in Fig. 3(a), gate x_4 at layer 2 connects with PI x_1 at the PI layer. To avoid the cross-layer connections in the reference parallel OIG architecture, we propose *implicit cross-layer connections* by adding buffers in the intermediate layers. Specifically, given two layers p and q $(q - p > 1)$, if there exists a cross-layer connection between node x_i at layer p and node x_k at layer q, we add a chain of $(q - p - 1)$ buffers after node x_i and connect the last buffer in the chain to node x_k. For example, Fig. 3(b) shows the architecture with implicit cross-layer connections for the reference parallel OIG architecture in Fig. 3(a).

After obtaining the architecture with implicit cross-layer connections, we obtain its gate number r, number of layers L, and number of nodes N_l at each layer l $(1 \leq l \leq L)$. Given the above parameters, we build the parallel OIG architecture. First, we renumber the nodes after performing a topological sort. Then, we establish the potential connections, where only the nodes of adjacent layers can connect with each other. For example, gate x_i at layer p can only take fanins from the nodes at layer $(p-1)$. Given the reference parallel OIG architecture without cross-layer connections shown in Fig. 3(b), Fig. 3(c) shows its corresponding parallel OIG architecture, where the dashed lines represent the possible connections between nodes.

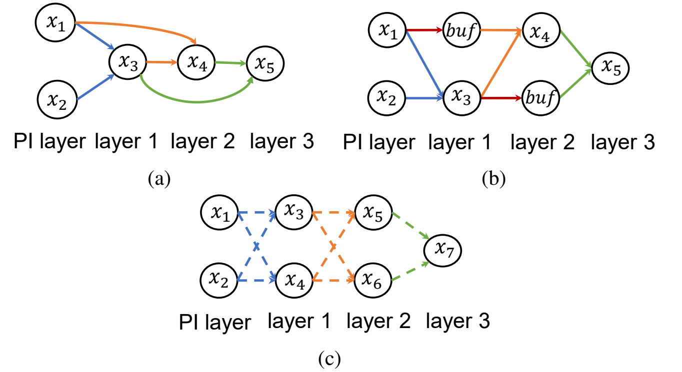

Fig. 3. (a) A reference parallel OIG architecture; (b) adding buffers to the reference parallel OIG architecture to realize cross-layer connections; (c) a parallel OIG architecture.

The method to obtain the candidate signals for the inputs of gate x_i in the parallel OIG architecture is similar to that in the serial one. The difference is that for gate x_i at layer p in the parallel OIG architecture, the total number of candidate fanin nodes of gate x_i, *i.e.*, K_i, equals the number of nodes at layer $(p - 1)$, and the candidate fanin node with the smallest ID is the first node at layer $(p - 1)$.

After obtaining the candidate signals for the input y_{ij} of gate x_i, y_{ij} is also computed by Eqs. (1)–(3). In addition, the output y_i of gate x_i is also calculated by Eq. (4).

Next, we also compute the number of connection indicator variables of a parallel OIG architecture. Let $N_{\max} = \max_{1 \leq l \leq L} N_l$. For each input y_{ij} of any gate x_i, the value K_i is bounded by $N_{\max}$. By Eq. (5), the number of connection indicator variables in the parallel OIG architecture is bounded by:

$$\sum_{i=n+1}^{n+r} 2 \times (2K_i + 1) \leq \sum_{i=n+1}^{n+r} 2 \times (2N_{\max} + 1) = 4rN_{\max} + 2r.$$

Thus, as the gate number r increases, the total number of connection indicator variables of a parallel OIG architecture grows linearly. Compared with the serial OIG architecture, the parallel one is more efficient for the architecture search of large circuits.

B. Continuous Relaxation of Search Space

In this work, we want to apply the gradient descent-based method to search the circuit architecture. However, in the search space, the connection indicator variables w_{ijk}'s in Eqs. (1)– (3) are discrete. In addition, the node output in Eq. (4) is calculated based on logical OR, which is also discrete. Thus, we make continuous relaxation of the connection indicator variables and node outputs, which is shown in Sections II-B1 and II-B2, respectively.

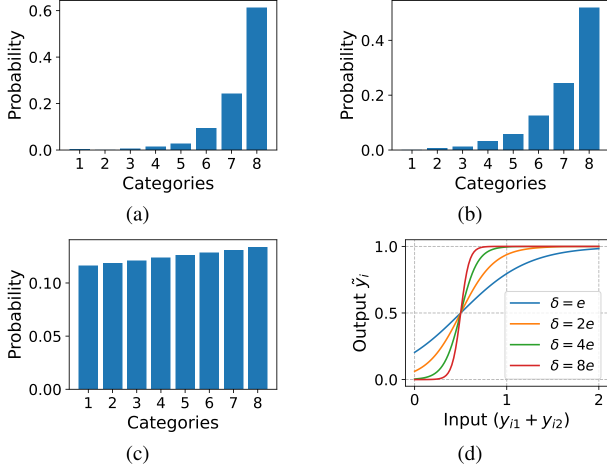

Fig. 4. Given $\boldsymbol{\theta_{ij}} = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]$, the distributions of Gumbel Softmax function with different temperature parameters τ: (a) $\tau = 0.1$; (b) $\tau = 1$; (c) $\tau = 50$. (d) The modified sigmoid functions $\tilde{y}_i$ with different parameters δ.

1) Continuous Relaxation of Connection Indicator Variable: By Eq. (3), among all connection indicator variables w_{ijk}'s of input j of gate x_i, only one is 1, and the others should be 0. In other words, the connection indicator vector $\mathbf{w}_{ij}$ is a *one-hot vector*.

To characterize this feature while also ensuring the continuity of w_{ijk}, we relax the discrete indicator variable w_{ijk} to be a continuous random variable computed by the Gumbel Softmax function [22]:

$$w_{ijk} = \text{GumbelSoftmax}(\theta_{ijk}|\boldsymbol{\theta_{ij}}) = \frac{e^{(\theta_{ijk}+\gamma\cdot g_{ijk})/\tau}}{\sum_k e^{(\theta_{ijk}+\gamma\cdot g_{ijk})/\tau}}, \quad (6)$$

where θ_{ijk} is a real weight parameter, g_{ijk} is a random noise following the Gumbel$(0, 1)$ distribution [22], γ is a scale parameter controlling the amplitude of noise g_{ijk}, and τ is a temperature parameter controlling the distribution of the Gumbel Softmax function. For $1 \leq k \leq 2K_i + 1$, parameters θ_{ijk}'s form a *real weight vector* $\boldsymbol{\theta_{ij}}$. For $n + 1 \leq i \leq n + r$, $1 \leq j \leq 2$, the vectors $\boldsymbol{\theta_{ij}}$ form a three-dimensional *real weight tensor* $\boldsymbol{\Theta}$, which represents the whole architecture of the OIG circuit *within the real weight space*. In this representation, w_{ijk} represents the *probability* that the input j of gate x_i connects with the candidate signal k. For the Gumbel Softmax function, as τ approaches 0, it approximates the discrete categorical sampling. As τ becomes larger, w_{ijk} becomes a continuous random variable. Figs. 4(a)– 4(c) illustrate Gumbel Softmax distributions with different temperature parameters τ. For $\tau = 0.1$ in Fig. 4(a), the Gumbel Softmax distribution approximates a discrete categorical distribution. For $\tau = 1$ in Fig. 4(b), the probability distribution becomes smoother compared to the case with $\tau = 0.1$, but still presents some level of discrepancy. For $\tau = 50$ in Fig. 4(c), the probabilities across categories are more evenly spread.

In some cases (*e.g.*, circuit evaluation), we still need to obtain the one-hot representation of the vector $\mathbf{w}_{ij}$ from the real weight vector $\boldsymbol{\theta}_{ij}$ to accurately characterize the node connections. It can be achieved with the following function:

$$w_{ijk} = \text{DeriveOneHot}(\boldsymbol{\theta}_{ij}) = \begin{cases} 1 & \text{if } k = \arg\max_k \boldsymbol{\theta}_{ij}, \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

2) Continuous Relaxation of Logical OR Operation: The node output shown in Eq. (4) realizes the 2-bit OR. We want to approximate it with a continuous arithmetic function. Inspired by the work [23], we design a modified version of the sigmoid function to approximate 2-bit OR as follows:

$$\tilde{y}_i = \sigma((y_{i1} + y_{i2} - 0.5)\cdot\delta) = \frac{1}{1 + e^{-((y_{i1}+y_{i2}-0.5))\cdot\delta}}, \quad (8)$$

where $\tilde{y}_i$ is the approximate output of 2-bit OR, and y_{i1} and y_{i2} are the two inputs of OR gate x_i. The function first adds y_{i1} and y_{i2}. From their sum, 0.5 is subtracted, which effectively shifts the point of maximum slope in the sigmoid function from 0 to 0.5. δ is a tunable scale parameter that controls the degree of approximation, and a larger value of δ leads to a better approximation of 2-bit OR. Fig. 4(d) shows the distribution of the modified sigmoid function $\tilde{y}_i$ with different choices of the parameter δ ranging from e to $8e$. The horizontal axis represents the sum of y_{i1} and y_{i2}, ranging from 0 to 2. When $y_{i1} + y_{i2}$ equals 0, 1, and 2, the ideal outputs are 0, 1, and 1, respectively. As δ increases, the output of $\tilde{y}_i$ approaches the ideal output more closely, indicating that a larger δ results in a better approximation of the ideal output.

C. Loss Function

The loss function has to reflect not only the error of a given circuit but also its hardware cost. Thus, we define the loss function as:

$$\mathcal{L}(ty, \widetilde{ty}, \mathbf{W}) = \alpha E(ty, \widetilde{ty}) + \beta C(\mathbf{W}). \quad (9)$$

The first term $E(ty, \widetilde{ty})$ is the circuit error, where ty and $\widetilde{ty}$ represent the outputs of the accurate and the approximate circuits, respectively. $E(ty, \widetilde{ty})$ can be any differentiable error metric, *e.g.*, MSE. In addition, the *normalized mean error distance (NMED)* and the *mean relative error distance (MRED)* are also applicable, where NMED measures the average *error distance (ED)* normalized by the maximum output value, while MRED measures the average relative ED [24]. These error metrics are non-differentiable only at zero, which can be effectively bypassed by setting the derivative at zero as zero. The second term $C(\mathbf{W})$ in Eq. (9) denotes the hardware cost of the circuit, where $\mathbf{W}$ is the connection indicator tensor. $C(\mathbf{W})$ can be any differentiable hardware cost metric. The parameters α and β are the weights that balance the importance of the error and the hardware cost.

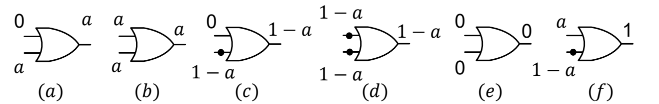

Fig. 5. Use an OR gate to realize (a) (b) buffer, (c) (d) inverter, (e) constant 0, and (f) constant 1.

The circuit error $E(ty, \widetilde{ty})$ can be easily modeled and computed. However, it is challenging to model a differentiable hardware cost term $C(\mathbf{W})$ with respect to $\mathbf{W}$. Here, we propose a solution to roughly estimate the circuit area of an OIG, and we use $C(\mathbf{W})$ to represent the circuit area. In fact, a node in an OIG can realize 5 gate types: OR gate, buffer, inverter, constant 0, and constant 1. Fig. 5 shows how to use OR gate to realize buffer, inverter, constant 0, and constant 1, where the black marker at an input of OR gate represents the logical negation. Given a signal a, a buffer can be realized in two ways: taking 0 and a as inputs (Fig. 5(a)), or both a as inputs

(Fig. 5(b)). Similarly, an inverter can also be realized in two ways as shown in Figs. 5(c) and (d). The constant 0 is realized by taking both 0 as inputs (Fig. 5(e)). The constant 1 is realized by taking a and $(1-a)$ as inputs (Fig. 5(f)). When performing technology mapping, the buffer, constant 0, and constant 1 in the OIG have negligible area. Thus, we assume that their areas are all 0. According to the Nangate 45nm standard-cell library [25], the area ratio between an inverter and an OR gate is approximately 1:2. Thus, we assume that the areas of an inverter and an OR gate are 1 and 2, respectively.

To estimate the circuit area $C(\mathbf{W})$ of an OIG, first we estimate the area for gate x_i in the OIG. During circuit training, we want $C(\mathbf{W})$ to be differentiable. Thus, $\mathbf{W}$ is a continuous connection indicator tensor obtained from Eq. (6). During circuit evaluation, we want to accurately characterize the discrete node connections and do not need $C(\mathbf{W})$ to be differentiable. Thus, $\mathbf{W}$ is the connection indicator tensor with one-hot representation obtained from Eq. (7).

After obtaining the indicator vector $\mathbf{w}_{ij}$ $(1 \le j \le 2)$, for each element w_{ijk} in $\mathbf{w}_{ij}$, it represents the probability that input j of gate x_i connects with the signal k $(1 \le k \le 2K_i+1)$. Then, we calculate the joint probability matrix $\mathbf{P}_i$ for gate x_i as:

$$\mathbf{P}_i = \mathbf{w}_{i1}^T \cdot \mathbf{w}_{i2}, \tag{10}$$

where $\mathbf{P}_i[k_1,k_2]$ $(1 \le k_1,k_2 \le 2K_i+1)$ represents the joint probability that the first and the second inputs of gate x_i connect with signals k_1 and k_2, respectively.

Example 3. *For the parallel OIG architecture shown in Fig. 3(c), we calculate the joint probability matrix $\mathbf{P}_3$ for gate x_3 at layer 1. Since there are 2 nodes in the PI layer, we have $K_3 = 2$. Thus, the number of candidate signals for an input of x_3 is $2K_3+1 = 5$. They are x_1, x_2, $1-x_1$, $1-x_2$, and constant 0, which are indexed as 1, 2, 3, 4, and 5, respectively. Assume that the connection indicator vectors for the first and the second inputs of x_3 are $\mathbf{w}_{31} = [0.02, 0.01, 0.9, 0.05, 0.02]$ and $\mathbf{w}_{32} = [0.05, 0.05, 0.02, 0.03, 0.85]$, respectively. Thus, the joint probability matrix $\mathbf{P}_3$ is calculated as:*

$$\begin{aligned}\mathbf{P}_3 &= \mathbf{w}_{31}^T \cdot \mathbf{w}_{32} \\ &= \begin{bmatrix} 0.0010 & 0.0010 & 0.0004 & 0.0006 & 0.0170 \\ 0.0005 & 0.0005 & 0.0002 & 0.0003 & 0.0085 \\ 0.0450 & 0.0450 & 0.0180 & 0.0270 & 0.7650 \\ 0.0025 & 0.0025 & 0.0010 & 0.0015 & 0.0425 \\ 0.0010 & 0.0010 & 0.0004 & 0.0006 & 0.0170 \end{bmatrix}.\end{aligned} \tag{11}$$

The maximum term in $\mathbf{P}_3$ is $\mathbf{P}_3[3,5] = 0.7650$, which means that the joint probability that the first and second inputs of x_3 connect with signal $(1-x_1)$ and constant 0, respectively, is 0.7650.

Then, we construct the *area reference matrix* $\mathbf{A}_i$ for gate x_i, where $A_i[k_1,k_2]$ $(1 \le k_1,k_2 \le 2K_i+1)$ represents the area of gate x_i if the first and the second inputs of gate x_i connect with signals k_1 and k_2, respectively. Note that from its two input signals, we can infer the gate type of x_i, and hence, its area.

Example 4. *For the parallel OIG architecture shown in Fig. 3(c), we calculate the area reference matrix $\mathbf{A}_3$ for gate x_3. The candidate signals for an input of x_3 are obtained from Example 3. If both inputs of gate x_3 are x_1, x_3 is a buffer. Since the index of x_1 is 1, we have $A_3[1,1] = 0$. If the two inputs of gate x_3 are $(1-x_1)$ and x_2, x_3 is an OR gate. Since the indices of $(1-x_1)$ and x_2 are 3 and 2, respectively, we have $A_3[3,2] = A_3[2,3] = 2$. The other terms in matrix $\mathbf{A}_3$ are calculated in a similar way. Finally,*

$$\mathbf{A}_3 = \begin{bmatrix} 0 & 2 & 0 & 2 & 0 \\ 2 & 0 & 2 & 0 & 0 \\ 0 & 2 & 1 & 2 & 1 \\ 2 & 0 & 2 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}. \tag{12}$$

Next, we calculate the area a_i of gate x_i based on the joint probability matrix $\mathbf{P}_i$ and the area reference matrix $\mathbf{A}_i$:

$$a_i = \sum_{k_1=1}^{2K_i+1} \sum_{k_2=1}^{2K_i+1} (\mathbf{P}_i \circ \mathbf{A}_i)[k_1,k_2], \tag{13}$$

where $\circ$ represents the element-wise (or Hadamard) product. It means that the area a_i is calculated by considering the probabilities of different gate types it can be and the corresponding areas of these gate types. For instance, for gate x_3 in the parallel OIG architecture shown in Fig. 3(c), based on Eqs. (11)–(13), its area can be calculated as $a_3 = \sum_{k_1=1}^{5}\sum_{k_2=1}^{5}(\mathbf{P}_3 \circ \mathbf{A}_3)[k_1,k_2] = 1.465$.

After obtaining the gate area a_i for each gate x_i, the total circuit area $C(\mathbf{W})$ is computed as:

$$\sum_{i=n+1}^{n+r} a_i = \sum_{i=n+1}^{n+r} \sum_{k_1=1}^{2K_i+1} \sum_{k_2=1}^{2K_i+1} (\mathbf{P}_i \circ \mathbf{A}_i)[k_1,k_2]. \tag{14}$$

D. Search Algorithm

The main procedure of DASALS is shown in Algorithm 1. We first discuss the basic idea of the procedure. DASALS is based on the mini-batch gradient descent method. Gradient descent begins with forward propagation, where input data is passed through the NN model to generate a prediction, and the loss of this prediction is calculated using a loss function. Then, in backward propagation, this loss is used to update the parameters of the model in the direction that minimizes the loss. Similar to this process, during each step of DASALS training, based on the continuous relaxation of the OIG architecture, we perform circuit forward propagation to calculate the differentiable loss $\mathcal{L}_s$ (Lines 9–11), which is used later to update the parameters of the OIG architecture during the backward propagation (Line 16). Typically, gradient descent is used in tasks where the calculated loss is based on the dataset's actual and predicted outputs (*e.g.*, linear regression, neural network training), thereby accurately reflecting the performance of the current model. However, for DASALS, gradient descent operates on the space that is only a continuous relaxation of the actual discrete OIG architecture. Thus, the differentiable loss $\mathcal{L}_s$ is just an approximation of the true loss. To accurately evaluate whether the current OIG architecture is a feasible solution, we need to determine the true loss of the discrete OIG architecture. Therefore, based on the discrete OIG architecture, another round of circuit forward propagation is needed to obtain the true loss $\mathcal{L}_h$ (Lines 12–14).

Now, we discuss the entire procedure in detail. Its inputs are an accurate circuit $\mathcal{G}_{acc}$ with PI number n, PO number m, and OIG node number r_{acc}, a reference approximate circuit $\mathcal{G}_{ref}$, an OIG node number bound r_{lim} used to determine whether to use a serial or a parallel OIG architecture, epoch number M, batch size bs, learning rate lr, temperature parameters τ_{fw} and τ_{bw}, scale parameters δ_{fw} and δ_{bw}, noise scale parameter γ, and loss bound $\mathcal{L}_{lim}$. The output of the algorithm is a set of approximate circuits $S_{\mathcal{G}}$ with the loss smaller than $\mathcal{L}_{lim}$.

In Algorithm 1, based on the OIG node number r_{acc}, Lines 1–3 determine to use the serial or the parallel OIG architecture. If r_{acc} is smaller than the OIG node number bound r_{lim}, Line 2 calls the function *BuildSerialOIGArchitect* to build the serial OIG architecture $\mathcal{D}$ based on the reference approximate circuit $\mathcal{G}_{ref}$. The details of the function are shown in Section II-A1. Otherwise, Line 3 calls the function *BuildParallelOIGArchitect* to build the parallel OIG architecture $\mathcal{D}$ based on $\mathcal{G}_{ref}$. The details of the function are shown in Section II-A2. Line 4 initializes the set $S_{\mathbf{W}}$ as an empty set, which is used to store the feasible connection indicator tensors with one-hot representation. Based on the reference approximate circuit $\mathcal{G}_{ref}$, Line 4 also initializes

the real weight tensor $\mathbf{\Theta}$ of the OIG architecture $\mathcal{D}$, which is the variable we want to train to minimize the loss function. The details of the initialization will be described in Section II-E1. By simulating the accurate circuit $\mathcal{G}_{acc}$, Line 5 calls the function *ObtainTrainingData* to obtain the training dataset $\mathbf{X}$ and $\mathbf{Y}$, which store the input patterns and the accurate outputs of $\mathcal{G}_{acc}$, respectively. Given the batch size bs, Line 6 calls the function *ObtainBatchData* to shuffle and divide the training dataset $\mathbf{X}$ and $\mathbf{Y}$ into a set of mini-batches $\mathbf{TX}$ and $\mathbf{TY}$.

Algorithm 1: The procedure of DASALS.

Input: an accurate circuit $\mathcal{G}_{acc}$ with PI number n, PO number m, and OIG node number r_{acc}, a reference approximate circuit $\mathcal{G}_{ref}$, an OIG node number bound r_{lim}, epoch number M, batch size bs, learning rate lr, temperature parameters τ_{fw} and τ_{bw}, scale parameters δ_{fw} and δ_{bw}, noise scale parameter γ, and loss bound $\mathcal{L}_{lim}$.
Output: a set of approximate circuits $S_{\mathcal{G}}$ with the loss smaller than $\mathcal{L}_{lim}$.

```
1  if r_acc < r_lim then
2  |  D ← BuildSerialOIGArchitect(n, m, G_ref);
3  else D ← BuildParallelOIGArchitect(n, m, G_ref);
4  S_W ← ∅; Initialize the real weight tensor Θ of D based on G_ref;
5  (X, Y) ← ObtainTrainingData(G_acc);
6  (TX, TY) ← ObtainBatchData(X, Y, bs);
7  for epoch i = 1 to M do
8  |  for each batch (tx, ty) ∈ (TX, TY) do
9  |  |  W_s ← GumbelSoftmax(Θ, γ, τ_fw);
10 |  |  ~ty_s ← CalCircuitSoftOutput(D, W_s, tx, δ_fw);
11 |  |  L_s ← CalculateLoss(ty, ~ty_s, W_s);
12 |  |  W_h ← DeriveOneHot(Θ);
13 |  |  ~TY ← CalCircuitHardOutput(D, W_h, TX);
14 |  |  L_h ← CalculateLoss(TY, ~TY, W_h);
15 |  |  if L_h < L_lim then S_W ← S_W ∪ W_h;
16 |  |  Θ ← BackwardPropagation(D, L_s, lr, τ_bw, δ_bw);
17 For the architectures D with the connection indicator tensors in
   S_W, perform logic synthesis and technology mapping to obtain a
   set of approximate circuits S_G;
18 return S_G
```

Lines 7–16 apply the mini-batch gradient descent method to search for the feasible OIG architecture. For each epoch, Line 8 iterates through each batch of input data tx and output data ty. First, Lines 9–11 perform the circuit forward propagation. They operate on the continuous relaxation of the OIG architecture $\mathcal{D}$, where all the operations are differentiable. Based on the real weight tensor $\mathbf{\Theta}$, noise scale parameter γ, and the temperature parameter τ_{fw}, Line 9 calls the function *GumbelSoftmax* in Eq. (6) to obtain the continuous connection indicator tensor $\mathbf{W}_s$, which is differentiable with respect to $\mathbf{\Theta}$. According to Figs. 4(a)–4(c), a small value τ_{fw} is chosen for the parameter τ in Eq. (6) to more closely approximate the actual one-hot node connections. Given the OIG architecture $\mathcal{D}$ with the continuous connection indicator tensor $\mathbf{W}_s$, the input batch data tx, and the sigmoid scale parameter δ_{fw}, Line 10 calls the function *CalCircuitSoftOutput* to calculate the approximate outputs $\widetilde{ty}_s$ of the OIG architecture. During this process, Eqs. (1) and (2) are applied to calculate the inputs of each gate, and Eq. (8) is used to calculate the output of each gate. According to Fig. 4(d), a large value δ_{fw} is chosen for the parameter δ in Eq. (8) to more closely approximate the 2-bit OR. Then, based on the accurate outputs ty, the approximate output $\widetilde{ty}_s$, and the continuous connection indicator tensor $\mathbf{W}_s$, Line 11 calls the function *CalculateLoss* to calculate the differentiable loss $\mathcal{L}_s$. The details of the function are shown in Section II-C. Lines 12–14 evaluate the circuit, which operates on the discrete OIG architecture $\mathcal{D}$. Circuit evaluation needs to accurately characterize the node connections and compute the gate output. Thus, based on the real weight tensor $\mathbf{\Theta}$, Line 12 calls the function *DeriveOneHot* in Eq. (7) to obtain the connection indicator tensor $\mathbf{W}_h$ with one-hot representation. Given the whole input dataset $\mathbf{TX}$ and the OIG architecture $\mathcal{D}$ with the connection indicator tensor $\mathbf{W}_h$, Line 13 calls the function *CalCircuitHardOutput* to calculate the set of all approximate outputs $\widetilde{\mathbf{TY}}$ of the OIG architecture. The detailed process is realized by Eqs. (1), (2), and (4). Similar to Line 11, Line 14 calls the function *CalculateLoss* to calculate the true loss $\mathcal{L}_h$. If the true loss $\mathcal{L}_h$ is smaller than the loss bound $\mathcal{L}_{lim}$, $\mathbf{W}_h$ with one-hot representation is a feasible architecture, and Line 15 adds $\mathbf{W}_h$ to set $S_{\mathbf{W}}$. Then, based on the OIG architecture $\mathcal{D}$, differentiable loss $\mathcal{L}_s$, learning rate lr, temperature parameter τ_{bw}, and scale parameter δ_{bw}, Line 16 calls the function *BackwardPropagation* to apply the gradient descent method to update the real weight tensor $\mathbf{\Theta}$. To avoid gradient vanishing for Gumbel Softmax function in Eq. (6) and sigmoid function in Eq. (8), we need to smooth the functions' curves. According to Fig. 4, a large value τ_{bw} is chosen for the parameter τ in Eq. (6), and a small value δ_{bw} is chosen for the parameter δ in Eq. (8). For the architectures $\mathcal{D}$ with the connection indicator tensors in $S_{\mathbf{W}}$, Line 17 performs logic synthesis and technology mapping to obtain a set of approximate circuits $S_{\mathcal{G}}$. Finally, Line 18 returns the set $S_{\mathcal{G}}$.

E. Implementation Tricks of DASALS

In this section, we present the implementation tricks of DASALS, making the architecture search computationally feasible and efficient.

1) Initializing from a Good Baseline: For the OIG architecture $\mathcal{D}$, proper initialization of the real weight tensor $\mathbf{\Theta}$ plays a crucial role. Our initialization strategy exploits the reference approximate circuit $\mathcal{G}_{ref}$. We first derive a connection indicator tensor $\mathbf{W}_{h0}$ with one-hot representation based on the connections of $\mathcal{G}_{ref}$, where the existence and the absence of a signal connection correspond to a 1 and a 0, respectively. Then, we obtain an initial real weight tensor $\mathbf{\Theta}_0$ based on $\mathbf{W}_{h0}$. To achieve this, we define an inverse mapping function to Eq.(7), *i.e.*, *OneHotToReal*:

$$\theta_{ijk} = \text{OneHotToReal}(\mathbf{w}_{ij}) = \begin{cases} T & \text{if } w_{ijk} = 1, \\ -T & \text{otherwise,} \end{cases} \quad (15)$$

where T denotes a real number greater than zero, serving as a tunable parameter. Given $\mathbf{W}_{h0}$ and applying Eq. (15), we can derive the initial real weight tensor $\mathbf{\Theta}_0$. This allows for a meaningful initialization of $\mathbf{\Theta}$ using $\mathbf{\Theta}_0$, thereby laying a solid foundation for the subsequent learning process of the OIG architecture.

2) Escaping from the Bad Local Minimum: In the training process, the algorithm may get stuck in a local optimum. To mitigate this issue, we propose to escape from the bad local minima by keeping track of the best state encountered during training and backtracking to that state if the algorithm finds itself in a worse position. Specifically, we maintain two key variables throughout the training process: a best connection indicator tensor with one-hot representation, denoted as $\mathbf{W}_{best}$, and the best true loss, $\mathcal{L}_{best}$. At the beginning, $\mathbf{W}_{best}$ is set as the initial connection indicator tensor with one-hot representation, while $\mathcal{L}_{best}$ is set as the initial true loss. During each step of training, we check if the current true loss $\mathcal{L}_h$ is less than $\mathcal{L}_{best}$. If so, it means that a better OIG architecture is found. Then, we update $\mathcal{L}_{best}$ to the current loss $\mathcal{L}_h$, and update $\mathbf{W}_{best}$ to the current connection indicator tensor $\mathbf{W}_h$. If the current true loss $\mathcal{L}_h > \mathcal{L}_{best} + \mathcal{L}_{ref}$, where $\mathcal{L}_{ref}$ is a predefined value, it means that the model is at a worse position in the search space, and thus, we backtrack. Specifically, given the value of $\mathbf{W}_{best}$, we apply the *OneHotToReal* function in Eq. (15) to obtain the value of corresponding $\mathbf{\Theta}$, and the current $\mathbf{\Theta}$ is then set to this value.

3) Avoiding Redundant Computation: Circuit evaluation in Lines 12–14 of Algorithm 1 is time-consuming, which requires a scan through the entire dataset. To manage this, we hash the already evaluated OIG architectures and their true losses, represented by the connection indicator tensor $\mathbf{W}_h$ and the true loss $\mathcal{L}_h$, respectively. Before starting an evaluation, we check if the current OIG architecture has been visited. If so, we skip the evaluation and directly retrieve the true loss, thus saving time. Otherwise, we perform the circuit evaluation and subsequently hash the architecture and its associated loss. This approach optimizes the evaluation process by avoiding redundant computation, improving the overall efficiency.

III. Experimental Results

This section shows the experimental results of DASALS. All the experiments of DASALS are conducted on an NVIDIA RTX 3090 GPU. All comparison methods are conducted on a 12-core Intel Xeon Gold 6146 processor using a single thread running at 3.2GHz with 62GB RAM. DASALS can handle any input distribution. In our experiments, the inputs are set as uniformly distributed. We use the MCNC library [26] as the technology library. Table I lists the benchmarks used in our experiments along with their IDs, node numbers in their OIG representations, areas, and delays. The circuits include some small arithmetic [8], large EPFL arithmetic [27], BACS [28], and LGSynt91 circuits [26]. The area and delay of each circuit are normalized to the area and delay of the INV_X1 gate in the MCNC library [26], respectively.

Table I. Benchmarks used in our experiments. #Nd: number of nodes.

Small arithmetic & EPFL arithmetic					BACS & LGSynt91 arithmetic				
ID	Ckt	#Nd	Area	Delay	ID	Ckt	#Nd	Area	Delay
1	alu4	1428	2408	28.4	8	abs_diff	104	212	25.8
2	cla32	420	796	38.2	9	alu2	361	622	46.8
3	ksa32	507	973	21.6	10	f51m	79	147	35
4	mtp8	515	917	41.6	11	mac	118	235	33.3
5	rca32	312	599	28.9	12	mult8	470	955	55.1
6	wal8	462	899	43.7	13	z4ml	34	60	9.6
7	sin	7044	12169	254.8	14	9symml	192	337	21

A. Parameter Setting

For circuit training of DASALS, as shown in Algorithm 1, Adam optimizer is used [29]. In Algorithm 1, we use an approximate circuit obtained from an existing ALS method, SEALS [8], under a randomly chosen error bound as the reference approximate circuit $\mathcal{G}_{ref}$. For the loss function in Eq. (9), the error term is computed using MSE, and the hardware cost function is based on the circuit area. In Eq. (9), the magnitude of the gradient of the error term is typically quite small, approximately in the order of 10^{-6}. To realize a proper trade-off between the error and the hardware cost, the parameters α and β are set as 1 and 10^{-6}, respectively. If the PI number n of the circuits is no more than 12, the total number of input patterns does not exceed 2^{12}. In this situation, we are not limited by the size of the GPU memory. Thus, the batch size bs and the epoch number M are set as 2^n and 100, respectively. Otherwise, due to GPU memory limit, we set $bs = 2048$ and $M = 20$. For the scale parameter δ_{fw} used in the circuit forward propagation, as the PI number n of the circuits increases, we need to increase δ_{fw} to more closely approximate the 2-bit OR. For the circuits with PI number n no more than 14, we set $\delta_{fw} = 4e$. For the circuits with PI number n between 15 and 20, we set $\delta_{fw} = 10e$. For the circuits with PI number n no less than 21, we set $\delta_{fw} = 20e$. To prevent gradient vanishing during the circuit backward propagation, the temperature parameter τ_{bw} for the Gumbel Softmax function is set to 1. In addition, the parameter γ in Eq. (6) is set as 0.5. The loss bound $\mathcal{L}_{lim}$ is set as the true loss of the reference circuit $\mathcal{G}_{ref}$. The parameter T in Eq. (15) is set as 5. The predefined loss $\mathcal{L}_{ref}$ in Section II-E2 is set as 50 times the true loss of the reference circuit $\mathcal{G}_{ref}$. The learning rate lr, the temperature parameter τ_{fw} used in the circuit forward propagation, and the scale parameter δ_{bw} used in the circuit backward propagation have a significant impact on the results. To mitigate randomness in the experiments, we use a fixed random seed and conduct a single run using this seed. Within this controlled setting, we explore a range of possible values for specific parameters. The learning rate lr is selected from the set $\{0.2, 0.4, 0.6, 0.8, 1, 5\}$, while τ_{fw} is chosen from $\{0.05, 0.1, 0.5, 1\}$, and δ_{bw} from $\{2e, 4e, 6e, 8e, 10e\}$. By iterating through these parameter combinations, we aim to identify the optimal configurations yielding the best ADP-accuracy trade-offs.

B. Comparison of the Serial and Parallel OIG Architectures of DASALS with MinAC

This section compares DASALS using the serial and the parallel architectures with a global ALS method, MinAC [13]. The benchmarks are *mul2* and *alu4*. *mul2* is a 2-bit multiplier synthesized using Yosys [30], while *alu4* is the circuit with ID 1 in Table I.

Table II. The MSE, hardware cost, and runtime comparison among MinAC and DASALS with the serial and parallel OIG architectures. N/A means that we cannot obtain the information.

Ckt	Method	MSE	Area	Delay	ADP	Runtime/min
mul2	MinAC	0.5	6	1.9	11.4	0.68
	DASALS-Serial	0.75	7	1.9	13.3	6.67
	DASALS-Parallel	1.25	7	2.5	17.5	1.96
alu4	MinAC	N/A	N/A	N/A	N/A	N/A
	DASALS-Serial	4498.88	17	5.4	91.8	79.14
	DASALS-Parallel	4529.73	18	6.4	115.2	15.10

Table II shows the MSE, hardware cost, and runtime comparison. For *mul2*, MinAC obtains the approximate circuit with the smallest MSE, area, delay, and ADP. It is followed by DASALS with the serial OIG architecture, and then DASALS with the parallel one. In terms of runtime, MinAC has the shortest runtime, followed by DASALS with the parallel OIG architecture, and finally DASALS with the serial one. This indicates that MinAC performs best for small circuits. For *alu4*, MinAC does not give a result in 12 hours due to the scalability issue. DASALS with the serial OIG architecture outperforms DASALS with the parallel architecture in terms of MSE, area, delay, and ADP. However, it runs $5.24\times$ slower. In general, DASALS addresses the scalability issue of MinAC. For DASALS, compared with the parallel architecture, the serial architecture offers a better trade-off between circuit quality and accuracy at the cost of longer runtime. Thus, in the following experiments, the OIG number bound r_{lim} is set as 200. If the OIG node number of the accurate circuit r_{acc} is smaller than r_{lim}, we use the serial OIG architecture. Otherwise, we use the parallel one.

C. Pareto Optimization of DASALS

In this section, we compare the approximate *alu4* circuits obtained by DASALS with those obtained by two state-of-the-art local ALS methods, SEALS [8] and BLASYS [9], under various values of MSE. To evaluate the synthesis quality of approximate circuits, we use *area ratio*, *delay ratio*, and *ADP ratio* (*i.e.*, the area/delay/ADP of the approximate circuit over that of the original one).

Fig. 6 shows the area/delay/ADP ratio-vs.-MSE plots for the approximate *alu4* circuits. The green line represents the Pareto-optimal frontier, where the Pareto-optimal points are all given by DASALS. Thus, the approximate *alu4* circuits obtained by DASALS outperform those of SEALS and BLASYS. More specifically, for the area ratio-MSE plot and the ADP ratio-MSE plot, the Pareto dominance relationship is as follows: DASALS performs the best,

followed by SEALS, and then BLASYS. Under similar levels of MSE, the area ratio and the ADP ratio for DASALS are up to 24.73% and 29.84% smaller than those of BLASYS, respectively. Similarly, the area ratio and the ADP ratio of DASALS are up to 2.08% and 12.42% smaller than those of SEALS, respectively. For the delay ratio-MSE plot, the Pareto dominance relationship shows that DASALS is superior, while SEALS and BLASYS have comparable performances. Under comparable levels of MSE, the delay ratio of DASALS can be up to 17.27% and 18.66% smaller compared to BLASYS and SEALS, respectively. Overall, DASALS can generate a wide range of Pareto-optimal approximate circuits with hardware cost-error trade-off, allowing users to choose based on their needs.

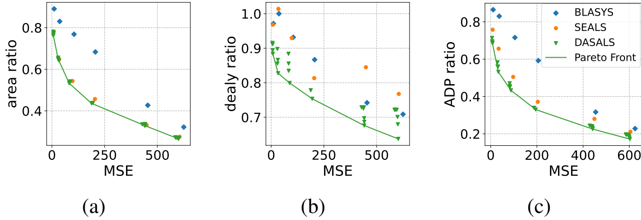

Fig. 6. The (a) area ratio-MSE, (b) delay ratio-MSE, and (c) ADP ratio-MSE, plots for the approximate *alu4* circuits.

Table III shows the average MSE and hardware cost comparison for *alu4* circuits under various values of MSE. On average, the area ratio, the delay ratio, and the ADP ratio of DASALS are 15.91%, 10.72%, and 19.52% smaller than those of BLASYS. When compared to SEALS, these ratios are 2.17%, 12.68%, and 7.74% smaller, respectively. At the same time, the approximate circuits obtained by DASALS exhibit the smallest MSE.

Table III. The average MSE and hardware cost comparison for *alu4* circuit.

Method	MSE	Area ratio	Delay ratio	ADP ratio
BLASYS	239.67	65.35%	87.01%	59.14%
SEALS	233.03	51.60%	88.97%	47.36%
DASALS	226.84	49.43%	76.29%	39.62%

D. Performance Comparison for Different Benchmarks

In this section, we compare DASALS with BLASYS and SEALS for a wide range of benchmarks. To evaluate the synthesis quality of approximate circuits, we use *relative MSE ratio*, *relative area ratio*, *relative delay ratio*, and *relative ADP ratio* (*i.e.*, the MSE/area/delay/ADP of the approximate circuit over that of SEALS).

Fig. 7 plots the relative MSE ratio, area ratio, delay ratio, and ADP ratio of the 14 benchmarks. For most benchmarks, the relative MSE ratio, area ratio, delay ratio, and ADP ratio of the approximate circuits obtained by DASALS are all better than or equal to those produced by BLASYS and SEALS. More specifically, the relative MSE ratio of DASALS are up to 73.08% (*z4ml* with ID 13) and 65.83% (*sin* with ID 7) smaller than those of BLASYS and SEALS, respectively. The relative area ratio of DASALS are up to 199.76% (*9symml* with ID 14) and 25% (*9symml* with ID 14) smaller than those of BLASYS and SEALS, respectively. The relative delay ratio of DASALS are up to 112.51% (*rca32* with ID 5) and 22.58% (*9symml* with ID 14) smaller than those of BLASYS and SEALS, respectively. The relative ADP ratio of DASALS are up to 379.68% (*9symml* with ID 14) and 41.94% (*9symml* with ID 14) smaller than those of BLASYS and SEALS, respectively.

Table IV shows the average error, hardware cost, and runtime comparison for all the benchmarks. On average, the relative MSE ratio, area ratio, delay ratio, and ADP ratio of DASALS are 16.61%, 28.66%, 41.73%, and 82.54% smaller than those of BLASYS. When

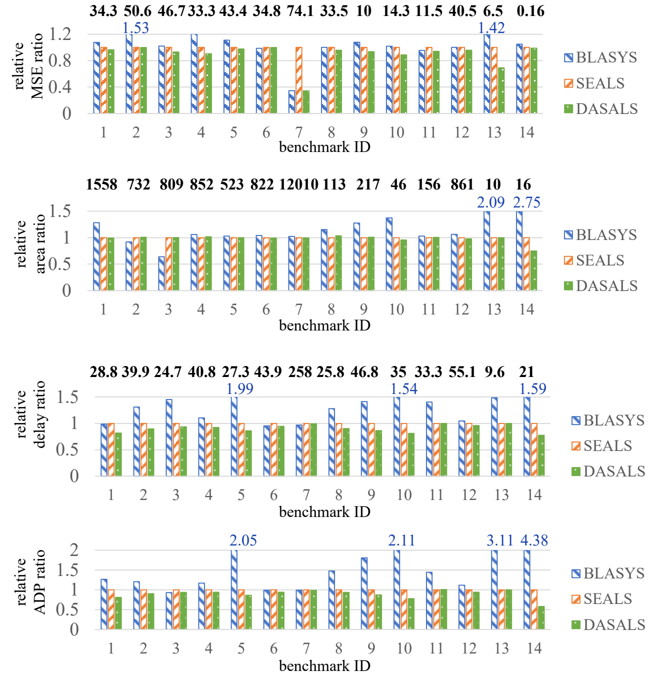

Fig. 7. The relative MSE ratio, area ratio, delay ratio, and ADP ratio of the 14 circuits. The MSE, area, and delay of the SEALS method are listed above each column in black. For the columns of BLASYS that exceed the maximum value of y-axis, the actual values are listed above these columns in blue.

Table IV. The average error, hardware cost, and runtime comparison for all benchmarks.

Method	MSE ratio	ER	Area ratio	Delay ratio	ADP ratio	Runtime/min
BLASYS	105.68%	77.54%	126.68%	132.28%	171.72%	13974.61
SEALS	100.00%	78.14%	100.00%	100.00%	100.00%	89.25
DASALS	89.07%	78.62%	98.02%	90.55%	89.18%	783.47

compared to SEALS, these ratios are 10.93%, 1.98%, 9.45%, and 10.82% smaller, respectively. In addition, there is negligible difference in the average *error rates (ERs)* among these three methods. For the runtime comparison, since DASALS uses the approximate circuit obtained from SEALS as a reference circuit, the runtime of DASALS consists of the runtime of SEALS plus the time DASALS itself takes to iterate through parameters. SEALS has the shortest runtime, followed by DASALS, and then BLASYS. Considering that DASALS produces a high-quality design, the longer runtime than that of SEALS is acceptable. Note that for DASALS, we iterate through 120 parameter combinations to obtain the best result. Thus, the average runtime of a single trial for DASALS is $\frac{783.47}{120} = 6.53$ min.

IV. Conclusion

In this work, we propose DASALS, a differentiable architecture search-driven ALS method to directly search the whole circuit structure to generate the approximate circuits with better circuit quality-accuracy trade-off. First, we model the discrete search space of ALS. Then, we propose methods to make continuous relaxation of the discrete search space. With its help, an efficient search algorithm is designed based on gradient descent. The experimental results show that compared with the state-of-the-art methods, DASALS significantly improves the quality of approximate circuits. Our future work will explore how to handle large-scale circuits and non-differentiable error metrics (*e.g.*, ER). We will also use parameter optimization techniques such as Bayesian optimization [31] and genetic algorithms [32] to quickly identify a good parameter combination for DASALS.

REFERENCES

[1] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, no. 7589, pp. 144–147, 2016.

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

[3] Q. Xu *et al.*, "Approximate computing: A survey," *IEEE Des. Test.*, vol. 33, no. 1, pp. 8–22, 2016.

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

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

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

[7] C. Meng *et al.*, "ALSRAC: Approximate logic synthesis by resubstitution with approximate care set," in *DAC*, 2020, pp. 1–6.

[8] C. Meng *et al.*, "SEALS: Sensitivity-driven efficient approximate logic synthesis," in *DAC*, 2022, pp. 439–444.

[9] S. Hashemi *et al.*, "BLASYS: Approximate logic synthesis using Boolean matrix factorization," in *DAC*, 2018, 55:1–55:6.

[10] M. Barbareschi *et al.*, "A catalog-based AIG-rewriting approach to the design of approximate components," *IEEE Trans. Emerg. Topics Comput.*, vol. 11, no. 1, pp. 70–81, 2023.

[11] S. Su *et al.*, "VECBEE: A versatile efficiency-accuracy configurable batch error estimation method for greedy approximate logic synthesis," *IEEE TCAD*, vol. 41, no. 11, pp. 5085–5099, 2022.

[12] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, 2014.

[13] X. Wang and W. Qian, "MinAC: Minimal-area approximate compressor design based on exact synthesis for approximate multipliers," in *ISCAS*, 2022, pp. 677–681.

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

[15] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," *ICLR*, 2019.

[16] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *ICLR*, 2017.

[17] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *ICLR*, 2017.

[18] E. Real, S. Moore, A. Selle, *et al.*, "Large-scale evolution of image classifiers," in *ICML*, 2017, pp. 2902–2911.

[19] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," *ICLR*, 2019.

[20] A. Biere, "The AIGER And-Inverter Graph (AIG) format version 20071012," *Tech. Rep.*, 2007.

[21] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, no. 02, pp. 107–116, 1998.

[22] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with Gumbel-softmax," *ICLR*, 2017.

[23] K. Hornik *et al.*, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.

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

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

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

[27] EPFL, *The EPFL combinational benchmark suite*, https://lsi.epfl.ch/page-102566-en-html/benchmarks/, 2021.

[28] I. Scarabottolo *et al.*, "Approximate logic synthesis: A survey," *Proc. IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015.

[30] C. Wolf., *Yosys Open SYnthesis Suite.* http://www.clifford.at/yosys/documentation.html, 2016.

[31] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *NIPS*, vol. 25, 2012.

[32] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.