Approximate Logic Synthesis for Dot-Inverter Graphs Using Node Merging-Enhanced Genetic Algorithm-Based Approach

Yi-Ting Li, Ihao Chen, Yung-Chih Chen, *Member, IEEE*, and Chun-Yao Wang, *Senior Member, IEEE*

***Abstract*— This article presents a novel approach to approximate logic synthesis (ALS) targeting at Dot-Inverter Graph (DIG), which is known for its superior expressive ability among all three-input gates and its potential in the future technology. We focus on minimizing the size of DIG circuits while maintaining acceptable error rates (ERs) by introducing a node merging (NM)-enhanced genetic algorithm (GA)-based approach. The NM technique reduces the DIG size without altering its functionality, while the GA, incorporating average relative Hamming distance (ARHD) and a self-adjusted mutation level (ML), is used for ALS on DIGs. Our experimental results demonstrated that the proposed approach achieves a higher reduction rate and less CPU time on different sizes of circuits compared to the state-of-the-art ALS approach.**

***Index Terms*— Approximate computing, circuit synthesis.**

I. Introduction

As THE technology node reaches its physical limitation, new devices are emerging to replace or supplement the existing CMOS technology potentially. One choice is the vertical nano-wire field-effect transistor (VNWFET), which is considered as a good candidate for advanced designs, and is efficient in three-input logic synthesis [7]. On the other hand, the Dot gate stands out in three-input logic gates for its effective expressiveness [29], meaning that Dot gates can implement the same function more efficiently compared with using other gates. Like {AND, INV} and {Majority, INV}, {Dot, INV} is also functionally complete, meaning that it can express all Boolean functions by combining members in the set. Marakkalage et al. [29] demonstrate that the Dot-Inverter Graph (DIG) representation has the smallest size when applied to EPFL benchmarks [5]. These developments motivate our exploration into logic synthesis to shrink the size of DIG-based circuits.

Recently, approximate circuits have gained significant attention [6], [19], [22], especially in applications such as image processing, where a certain degree of error can be tolerated. Approximate logic synthesis (ALS) aims to tradeoff accuracy against reductions in area, delay, or power consumption, making it a valuable approach for designing more efficient circuits. It allows for the synthesized circuit to have different output values from the original ones within an acceptable error margin. Given the potential of DIGs in minimizing circuit size, our focus shifts to applying ALS to further optimize these circuits. By allowing controlled errors, we can achieve more compact DIG-based circuits, making them suitable for resource-constrained environments.

Existing studies have demonstrated the effectiveness of approximate circuits. Some research is applicable in specific contexts, such as arithmetic circuits [19], [22], [23], [27], [55]. They typically operate at a higher level that is accessible to designers. This allows for manual execution and offers scalability for larger circuits. Other studies focused on logic rewriting methods operated based on the functionality of the circuit. These methods are performed without knowing the actual circuit implementation. By understanding a circuit's functionality, do not-cares can be extracted and then used for ALS [35]. Binary decision diagram (BDD)-based synthesis methods [14], [41] used the graph structure of BDDs to detect and utilize do not-cares for circuit simplification. However, as these methods only operate at the Boolean level, the size of the circuit is determined by the actual implementation. Hence, an iterative process of finding the smallest implementation is necessary after performing these methods.

Apart from Boolean-level operations, some studies worked on the gate-level representation, which allows for a more precise assessment of the final outcome of an approximate circuit. These studies typically involved structural analysis and pruning, resulting in a smaller search space compared to rewriting. However, it could also limit the potential of finding the best approximate circuit. Venkataramani et al. [47] proposed a substitution method to merge pairs of signals in the circuit that are likely to have the same function for approximation. Meng et al. [31] and Schlachter et al. [38]

Received 24 October 2024; revised 24 February 2025 and 11 June 2025; accepted 9 August 2025. Date of publication 27 August 2025; date of current version 23 March 2026. This work was supported in part by the Ministry of Science and Technology (MOST) under Grant MOST 111-2221-E-007-121 and Grant MOST 111-2221-E-011-137-MY3; in part by the National Science and Technology Council (NSTC) under Grant NSTC 112-2218-E-007-014, Grant NSTC 112-2221-E-007-106-MY2, Grant NSTC 112-2221-E-007-108, Grant NSTC 112-2425-H-007-002, Grant NSTC 113-2425-H-007-004, Grant NSTC 113-2640-E-011-003, Grant NSTC 113-2221-E-007-082-MY3, Grant NSTC 114-2221-E-007-125-MY3, Grant NSTC 114-2218-E-007-002, Grant NSTC 114-2221-E-007-126-MY3, and Grant NSTC 114-2425-H-007-002; and in part by the National Tsing Hua University (NTHU) under Grant NTHU 110A0119EX, Grant NTHU 111A0131EX, and Grant NTHU 112A0241EX. This article was recommended by Associate Editor E. Testa. *(Corresponding author: Yi-Ting Li.)*

Yi-Ting Li and Chun-Yao Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu 300044, Taiwan (e-mail: yitingli.yt@gmail.com; wcyao@cs.nthu.edu.tw).

Ihao Chen is with Incentia Design Systems Inc., Santa Clara, CA 95054 USA (e-mail: ihao@incentia.com).

Yung-Chih Chen is with the Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei 106335, Taiwan (e-mail: ycchen.ee@mail.ntust.edu.tw).

Digital Object Identifier 10.1109/TCAD.2025.3603509

introduced a structural pruning method by using each node's significance and activity in the circuit. Shin and Gupta [40] used stuck-at fault injection to simplify circuits. Tam et al. [44] merged gates in the circuits by relaxing conditions for logic substitution. Wu and Qian [50] simplified circuits by removing a few literals from the gates' expression in the circuits and checking if the resultant error is within constraints. Su et al. [42], [43] then advanced an error estimation method using Monte Carlo simulation and logic implication to forecast circuit errors before applying approximate changes.

However, the previous works apply only one approximate change per iteration, and operations are greedy. In other words, they select the operation that maximally reduces the circuit size in each iteration without considering the global context. To ensure that the selected approximate changes do not violate given constraints, these methods often involve evaluating each approximate change in every iteration and selecting the best one, which can be time-consuming. To address this, Meng et al. [32] proposed a faster method for evaluating approximate changes using sensitivity. In addition, to enable the application of multiple approximate changes in a single iteration, Wang et al. [49] introduced techniques to efficiently predict the overall impact of a set of approximate changes. However, due to the limitations of their prediction technique, they only considered approximate changes without structural dependencies in a single iteration. More recently, Meng et al. [30] proposed a resubstitution-based ALS framework. Their method introduced approximate changes by replacing a target node's fanins with alternative logic derived from other nodes, and employed random simulation to efficiently search the candidate nodes. To further reduce runtime, a knapsack-based selection strategy, together with a proposed error model, is used to apply multiple approximate changes per iteration. To reduce the search space, they restricted candidate nodes to be within the transitive fan-in (TFI) cone of the target node. While their methods can be used for and-inverter graphs (AIGs), it is not applicable to DIGs.

Recently, several studies have explored the use of evolutionary algorithms for ALS [17], [25], [34], [39], [45], [46]. Vasicek and Sekanina [45] demonstrated the effectiveness of Cartesian genetic programming (CGP) for circuit optimization. Hrbacek et al. [17] extended Vasicek and Sekanina [45] to the design of multiobjective approximate arithmetic circuits, incorporating tradeoffs among accuracy, power, and area. Subsequently, Mrazek et al. [34] developed the EvoApprox8b library, which provides a Pareto-optimal set of approximate 8-bit adders and multipliers, enabling flexible subcircuit replacement across multiple objectives such as power consumption, circuit area, and computational accuracy. However, these methods have been predominantly applied to arithmetic circuits, where the structured nature of arithmetic operations aligns well with CGP's encoding scheme. For nonarithmetic circuits, the CGP's representation becomes less effective.

Genetic algorithm (GA) [16] is a variant of evolutionary algorithms [13] and is an adaptive search algorithm commonly used to find optimal solutions [11]. Unlike traditional methods, GA is often applied to large search spaces to reach the global optimum gradually. As compared to other algorithms like simulated annealing [4], [20], which only performs single-point searches, GA allows for multiple-point searches, enabling exploration of different solutions and opportunities of escaping the local optimum. When applied to logic synthesis, GA can evaluate multiple changes in the circuit simultaneously, and its inherent random perturbation mechanism helps explore solution spaces that are inaccessible to greedy algorithms. Previous studies had applied GA to circuit optimization, such as reducing circuit area without changing the circuit's functionality [18], or reducing model size of AI applications [52], [54]. By relaxing the accuracy requirements, GA has also been found to be useful for approximate circuit synthesis [25], [46]. However, most of these studies only focused on small benchmarks or the circuits represented by AIGs. Therefore, we employ GA into our ALS approach for DIGs.

One of the key operations in GA is mutation, where the encoding of a circuit is altered so that some nodes in the circuit are removed. Instead of randomly removing nodes, it can sometimes be beneficial to guide the algorithm using logic substitution techniques, which can yield better circuit outcomes. Node merging (NM) is one such node removal technique used for gate-level circuit simplification, allowing for the removal of nodes while preserving the overall functionality of the circuit. This technique involves selecting a target node, n_t, in the circuit and search for a substitute node, n_s, to replace it. After this substitution, n_t and the nodes in its maximum fan-out-free cone (MFFC) can be removed from the circuit. An NM technique [9], [10] was introduced to simplify AIGs using mandatory assignments (MAs) for stuck-at faults. Subsequently, NM was also applied to circuit simplification for majority-inverter graphs (MIGs) [21]. The computation of MAs involves assigning noncontrolling values to propagate the stuck-at fault effect: in AIGs, the noncontrolling value for an AND gate is 1, and in MIGs, the noncontrolling pair for a three-input majority gate is achieved by assigning different values to its side-inputs. However, DIGs do not possess these properties. Thus, in this article, we propose a novel NM technique that can be applied directly to DIGs and can be leveraged to enhance the performance of our GA-based ALS. The contributions of this article are threefold.

1) We propose an NM algorithm used for DIG circuits.
2) We propose a GA-based ALS approach for DIG circuits.
3) The proposed algorithms can be integrated together, and are scalable for larger circuits.

II. Preliminaries

A. Backgrounds

The *Dot* function is an asymmetric three-input function, whose function is defined as $\mathrm{Dot}(x, y, z) = x \oplus (z + x \cdot y)$. A DIG is a directed acyclic graph consisting of only Dot gates and inverters represented as black nodes on edges. One example is shown in Fig. 1(a). We also use cycles to represent Dot gates in the DIG throughout this article. In our DIG representation as shown in Fig. 1(b), the fanins of a node are x, y, and z, respectively, from the top to the bottom of a cycle, which are not marked explicitly. The size of a DIG refers to the number of Dot gates in the DIG.

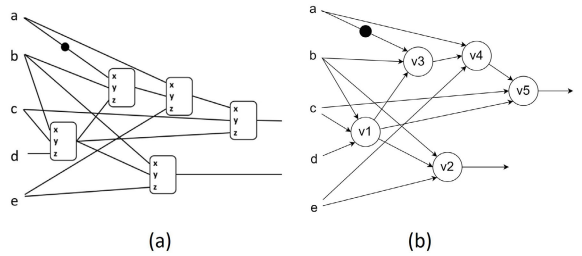

Fig. 1. Example of DIG: its (a) original form and (b) simplified form.

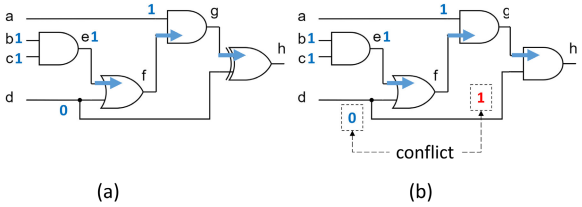

Fig. 2. Computation of MAs. (a) Computation of MAs(e = sa0), where the arrows indicate the fault-propagating path. (b) Inconsistent values in MAs(e = sa0).

An *input-controlling value* of a gate g is the input value that can determine the output value of g. An *input-noncontrolling value* is the input value that cannot determine the output value of g. For instance, the input-controlling value of an OR gate is 1, and its input-noncontrolling value is 0. A *dominator* of a gate g is a gate that must be passed from g to any primary outputs (POs). A side-input of a dominator d of g is a fan-in gate of d that is not in the transitive fan-out (TFO) cone of g.

A *stuck-at fault* is a fault model representing a manufacturing defect on a wire or a gate, which expresses the fault as a fixed value of 1 or 0, denoted as sa1 or sa0, respectively. A *stuck-at fault test* is a process of generating input patterns such that the output values of the faulty and fault-free circuits are different under these input patterns. To detect a stuck-at fault in the circuit, there must exist value assignments that activate and propagate the fault effect to any POs; otherwise, the fault is untestable. An untestable stuck-at fault implies that the fault site of the circuit can be replaced by the faulty value. This is because this faulty value cannot be observed at any POs.

The set of MAs for a stuck-at v (sav) fault test on a node n is denoted as MAs(n = sav). This represents a set of unique value assignments that activate and propagate the fault effect, and the value assignments that are inferred from them using logic implication.

We use the circuit in Fig. 2(a) to explain how MAs are calculated in a logic circuit. Suppose we want to compute MAs(e = sa0). First, e should be assigned the value for activating the fault effect, which is 1. The fault effect is then propagated along the path, including an OR gate, an AND gate, and an XOR gate. To propagate the fault effect, the side-inputs of the OR and AND gates need to be assigned noncontrolling values, which are $d = 0$ and $a = 1$, respectively. The XOR gate can propagate the fault effect regardless of its side-input value. Finally, based on logic implication, we can further deduce $(b, c) = (1, 1)$ from $e = 1$. Therefore, MAs(e = sa0) = $\{a = 1, b = 1, c = 1, d = 0, e = 1\}$. During the process of calculating MAs, conflicts may occur, i.e., the value assignments are inconsistent. For example, consider another circuit in Fig. 2(b). To propagate the fault effect from g to h, the side-input d of AND gate has to be assigned 1, which leads to a conflict with $d = 0$. Thus, MAs(e = sa0) do not exist, meaning that the sa0 fault at node e is an untestable fault.

B. Node Merging

NM determines if two nodes in a circuit can be merged or not without affecting the overall functionality of the circuit. When two nodes can be merged, that means that the error produced by merging of these nodes cannot be observed at any PO under all the input patterns. The method first selects a target node n_t in the circuit, then searches for another substitute node n_s satisfying a sufficient condition: Condition 1. If a node satisfies Condition 1, it is considered as n_s and can be used to replace n_t. As a result, the resultant circuit is minimized.

Condition 1 ([9], [10]): If $n_s = 1$ and $n_s = 0$ are MAs for stuck-at 0 and stuck-at 1 fault tests on n_t, respectively, n_s ($\overline{n_s}$) is a substitute node of n_t. By contrast, if $n_s = 0$ and $n_s = 1$ are MAs for stuck-at 0 and stuck-at 1 fault tests on n_t, respectively, $\overline{n_s}$ is a substitute node of n_t.

By Condition 1, we know that if we can find a node with different values in MAs(n_t = sa0) and MAs(n_t = sa1), respectively, this node is an n_s, and the error produced by the replacement of n_t with n_s cannot be observed at any POs. Thus, n_t can be replaced by n_s. The details of our NM on DIGs will be discussed in Section III.

C. Genetic Algorithm

Similar to how artificial neural networks mimic the neural networks in a brain, GAs aim to enhance the quality of a *population*, typically represents a set of solutions. It emulates the principles of genetic diversity and survival of the fittest individuals in the biological world. In GAs, the smallest unit is a *gene*, and the combination of multiple genes is referred to as a *chromosome*. Chromosomes determine the characteristics of an *individual*, which represents a single solution.

To maintain genetic diversity, GAs select two individuals from the population to serve as *parents*. The parents undergo *crossover* and *mutation* operations to produce new individuals, or called *offspring*. These offspring are added into the population. We apply *selection* operation with a fitness function, where individuals with higher fitness values are chosen to be parents for the next generation. This operation facilitates the production of fitter offspring. Crossover, mutation, and selection operations continue until the population is stabilized.

D. Multiobjective Optimization

In most synthesis problems, there exist tradeoffs between multiple objectives, such as increasing area for reducing delay, or reducing area at the expense of lower accuracy (i.e., larger error). The optimal solution to this problem is often referred to as a knee solution because it is located in the knee area of the Pareto front [51], [54]. As illustrated in Fig. 3(a), point S represents the knee solution. Reducing the area at this knee solution leads to a significant degradation in accuracy,

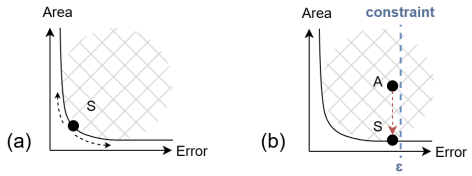

Fig. 3. Illustration of two different types of multiobjective optimization, where the cross-hatched area represents the solution space, the curved line represents the Pareto front, and the points, S and A, represent solutions. (a) Knee type. (b) Constrained type.

while reducing the error at this point results in a substantial increase in area. However, in our ALS problem, it is more of a constrained multiobjective problem, where we aim to find the minimum area within a given error constraint. Therefore, the optimal solution is not in the knee area but in the region of the Pareto front closest to the constraint, as illustrated by point S in Fig. 3(b).

E. Error Evaluation

In the ALS, it is crucial to evaluate the circuit's quality resulting from approximation. Various evaluation metrics have been proposed.

Error distance (ED) or error magnitude [26], [33], [36] represents the numerical difference that arises due to errors between the approximate circuit and the original one. ED is commonly used in the evaluation of approximate arithmetic circuits, such as adders, multipliers, etc. When using this error evaluation metric, it is necessary to define the representation of POs, such as the most significant bit (MSB) and the least significant bit (LSB). Normalized mean ED (NMED) represents the average ED normalized by the maximum output value, which is defined as follows:

$$\text{NMED} = \sum_{x \in X} \frac{|v(x) - v'(x)|}{2^{|PO|} - 1} \cdot \frac{1}{|X|} \tag{1}$$

where $v(x)$ and $v'(x)$ denote the numerical value at the outputs of the original circuit and the approximate one, respectively, when a specific input pattern x is applied. $|v(x) - v'(x)|$ represents the difference between their numerical values. $|X|$ represents the number of input patterns in X. |PO| is the number of POs in the circuit.

Hamming distance [28], [37], [48] measures the number of bit flips that occurs at the POs due to errors in the approximate circuit. However, the more common error evaluation metric is the error rate (ER) [12], [19], [22], [24], [25], [44], which is formulated as follows:

$$\text{ER} = \frac{|X_{\text{incorrect}}|}{|X|} \times 100\% \tag{2}$$

where $|X_{\text{incorrect}}|$ represents the number of simulation patterns in X that results in incorrect values at any POs.

In addition, to assess the number of POs affected by errors, we also utilize another metric, average relative Hamming distance (ARHD), to guide the evolution in GAs. ARHD is calculated as follows:

$$\text{ARHD} = \sum_{x \in X} \frac{d(f(x) - f'(x))}{|\text{PO}|} \cdot \frac{1}{|X|} \tag{3}$$

where $d(f(x) - f'(x))$ represents the Hamming distance between the approximate circuit and the original one at the POs when a specific input pattern x is applied. ARHD assigns a weight to each erroneous input pattern. This weight represents the proportion of POs that produce incorrect values under that input pattern. For example, suppose an approximate circuit C_1 has eight input patterns in total, and two of these patterns are erroneous. The first erroneous pattern causes (1/2) of the POs to produce incorrect values, and the second erroneous pattern causes (1/3) of the POs to produce incorrect values. The ARHD of this approximate circuit C_1 will be $[(1/2) + (1/3)] \times (1/8) = 10.42\%$. Now, consider another approximate circuit C_2, which also has two erroneous patterns. These erroneous patterns cause (1/5) and (1/10) of the POs to produce incorrect values, respectively. The ARHD for the circuit C_2 will be $[(1/5) + (1/10)] \times (1/8) = 3.75\%$. Hence, we consider the circuit C_2 as a better approximate circuit.

III. NM on DIGs

A. MA Computation for DIGs

The NM algorithm requires the computation of MAs for a target node n_t. MAs(n_t = sa1) and MAs(n_t = sa0) in a DIG are the fault-activating value on n_t, the fault-propagating values at side-inputs of n_t's dominators, and the values inferred by logic implication from these MAs. Here, we first focus on the generation of fault-activating value and the fault-propagating values of a Dot gate in DIGs.

To activate the fault effect on n_t, we have to assign a fault-free value at the fault site, i.e., assigning $n_t = 1$ and $n_t = 0$ for $n_t$sa0 fault, and $n_t$sa1 fault, respectively. To propagate the fault effect, we have to consider the combinations of value assignments at the side-inputs of dominators of n_t such that the fault effect can be propagated to any PO. In an AIG, the fault-propagating values are determined by assigning the noncontrolling value on all the side-inputs of the dominators of n_t. However, in a DIG, we need to consider a *noncontrolling pair*, in which a fault effect can be passed from a Dot gate's fan-in to its fan-out, at the side-inputs of each dominator of n_t. Furthermore, since the function of Dot gate is asymmetric, we need to consider all the cases to propagate the fault effect to the POs as summarized as follows: 1) the fault effect is at the input x; 2) the fault effect is at the input y; 3) the fault effect is at the input z; 4) the fault effects are at the inputs x and y; 5) the fault effects are at the inputs y and z; 6) the fault effects are at the inputs x and z; and 7) the fault effects are at all the inputs.

In the following paragraphs, we will explain the fault effect propagation of a Dot gate for different cases.

Case (1): Without loss of generality, we assume that the fault effect is 1/0, and is propagated from other nodes in the DIG. The side-inputs are input y and input z. According to the functionality of a Dot gate, we can see that x is a fan-out and there is an XOR at the end, as shown in Fig. 4(a). If the fault effects from x reach at both inputs of the XOR, the fault effects will be canceled out. Hence, by assigning a controlling value to the AND gate ($y = 0$) or to the OR gate ($z = 1$), we can block the fault effect at the second input of

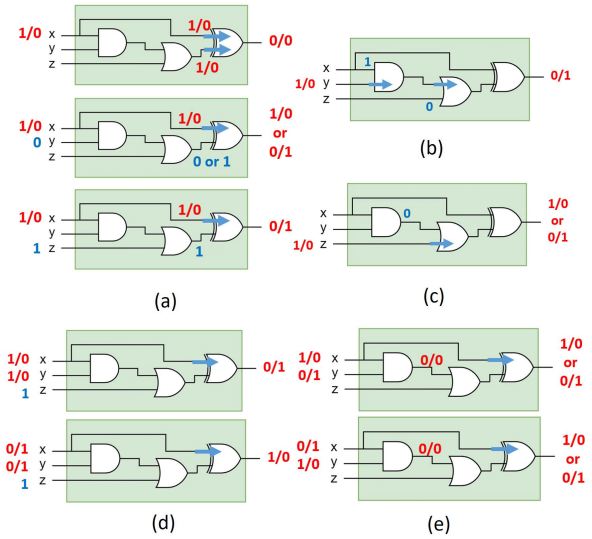

Fig. 4. Derivation of the noncontrolling pairs of a Dot gate. $1/0$ denotes that the fault-free value is 1, and the faulty value is 0. $1/0$ at the output represents a testable fault while $0/0$ represents an untestable fault. The arrows refer to the fault-propagating paths. (a) Case (1). (b) Case (2). (c) Case (3). (d) Case (4): identical fault effects are at the inputs x and y. (e) Case (4): opposite fault effects are at the inputs x and y.

TABLE I
NONCONTROLLING CONDITIONS FOR CASES (1) TO (7)

Case	Fault effect locations and types			Non-controlling cond.
	x	y	z	
(1)	D			$y = 0$ or $z = 1$
(2)		D		$x = 1$ and $z = 0$
(3)			D	$x = 0$ or $y = 0$
(4)	D	D		$z = 1$
	D	D′		∅
(5)		D	D	∅
		D	D′	$x = 0$
(6)	D		D	fault effects are blocked
	D		D′	$y = 1$
(7)	D	D	D′	∅
	others			fault effects are blocked

the XOR, as shown in the second and third circuits of Fig. 4(a). Therefore, the noncontrolling pair (y, z) of Case (1) is one of the three combinations: $\{(0, 0), (0, 1), (1, 1)\}$ for fault effect propagation.

Case (2): The side-inputs are input x and input z. To propagate the fault effect, the AND and OR along the propagating path should be assigned noncontrolling values, as shown in Fig. 4(b). Hence, the noncontrolling pair (x, z) of Case (2) is only $(1, 0)$.

Case (3): The value assignment is shown in Fig. 4(c). Hence, the noncontrolling pair (x, y) of Case (3) is one of the combinations: $\{(0, 0), (0, 1), (1, 0)\}$.

In addition to the cases that the fault effect is propagated from one input of a node, the cases that the fault effect is propagated from more than one input have to be considered. For these cases, we have to consider whether the fault effects at the inputs of a node are identical or opposite.

Case (4): In Fig. 4(d), identical fault effects are propagated to the input x and input y of a node simultaneously. Since the fault effects are identical, the side-input z has to be assigned 1 to block the fault effect from the second input of XOR, and propagate the other fault effect to the output. Therefore, the noncontrolling value in Case (4) is $z = 1$. However, when the fault effects are opposite, as shown in Fig. 4(e), one fault effect is propagated to the output under any z value. Therefore, there is no noncontrolling value in Case (4).

The noncontrolling conditions including Cases (5) to (7) are summarized in Table I, where "D" represents a fault effect, i.e., $1/0$ or $0/1$, at the side-inputs of the dominators of n_t and "D′" represents a negated fault effect of D. It is worth mentioning that the "fault effects are blocked" under the Column "Noncontrolling cond." in Table I, indicates that the fault effects will never be propagated to the output, which also means that n_t can be replaced with the faulty value.

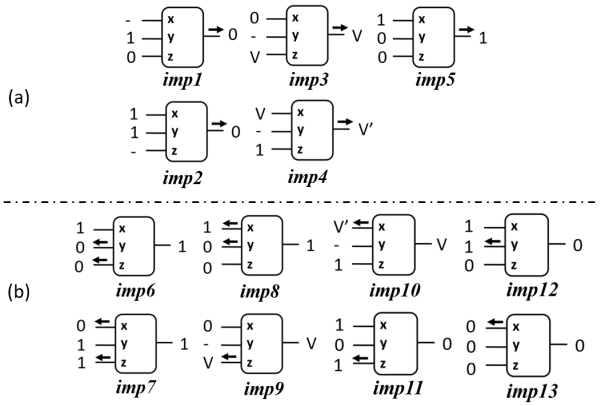

Fig. 5. Logic implication of a Dot gate, where "-" refers to do not care, which includes unknown value, "V" refers to a known value, which is either 1 or 0, and "→" and "←" refer to logic implications. (a) Forward cases. (b) Backward cases.

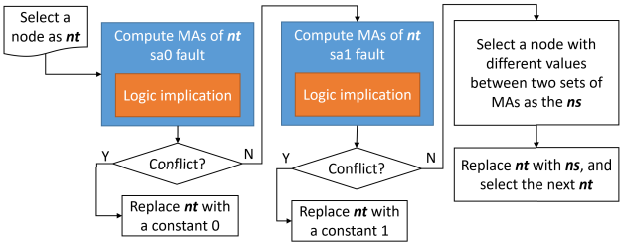

Fig. 6. Overall flow of proposed NM on DIGs.

B. Logic Implications for a Dot Gate

After assigning the fault-activating and fault-propagating values as MAs, we use logic implications to infer more MAs. Here, we try to find the sufficient conditions of the logic value combination to deduce other unknown values within the circuit. The logic value implications for a Dot gate are summarized in Fig. 5. When the value at the output can be inferred based on known values at the inputs, we term these scenarios as forward cases, as indicated as *imp1* to *imp5* in Fig. 5(a). For example, the *imp1* in Fig. 5(a) shows that $(y, z) = (1, 0)$ implies the output to be 0 without considering the value of input x due to $\text{Dot}(x, y, z) = x \oplus (z + x \cdot y) = x \oplus x = 0$. Conversely, when the inputs can be implied from the known value at the output, we term them as backward cases, as indicated as *imp6* to *imp13* in Fig. 5(b). For example, the *imp9* in Fig. 5(b) shows that $x = 0$ and any known output value imply that z must be the same value as the output, regardless of the value of input y.

TABLE II
DERIVATION STEPS OF LOGIC IMPLICATION FOR n_t = SA0

Side-input of v_4	Side-input of v_5	Step	Known value	Implied value	Rule
$a=1$ $e=0$	$c=0$ $v_1=0$	(1)	$a=1$, $v_3=1$	$v_4=0$	*imp2*
		(2)	$v_4=0$, $v_1=0$	$v_5=0$	*imp3*
		(3)	$\overline{a}=0$, $v_1=0$	$v_3=\underline{0}$	*imp3*
$a=1$ $e=0$	$c=0$ $v_1=1$	(1)	$a=1$, $v_3=1$	$v_4=0$	*imp2*
		(2)	$v_4=0$, $v_1=1$	$v_5=1$	*imp3*
		(3)	$v_1=1$, $e=0$	$v_2=0$	*imp1*
		(4)	$\overline{a}=0$, $v_1=1$	$v_3=1$	*imp3*
$a=1$ $e=0$	$c=1$ $v_1=1$	(1)	$a=1$, $v_3=1$	$v_4=0$	*imp2*
		(2)	$v_4=0$, $v_1=1$	$v_5=1$	*imp3*
		(3)	$v_1=1$, $e=0$	$v_2=0$	*imp1*
		(4)	$\overline{a}=0$, $v_1=1$	$v_3=1$	*imp3*
		(5)	$c=1$, $v_1=1$	$b=0$, $d=1$	*imp7*

TABLE III
THREE SETS OF VALUE ASSIGNMENTS ABOUT n_t = SA0

Side-input of v_4		**Side-input of v_5**		n_t	**Implied value**					**Conflict**
a	e	c	v_1	v_3	v_4	v_5	v_2	b	d	v_3
1	0	0	0	1	0	0				$\underline{0}$
1	0	0	1	1	0	1	0			1
1	0	1	1	1	0	1	0	0	1	1

In Section III-A, we have demonstrated that there may be multiple sets of fault-propagating values to propagate a fault effect due to multiple combinations of the values of the side-inputs. Hence, the sets of value assignments produced by logic implication with a fault-activating value and different combinations of fault-propagating values are intersected to deduce actual MAs.

C. Overall Flow of the NM

The overall flow of the proposed NM technique in a DIG is shown in Fig. 6. Given a node in the circuit as the target node n_t, we calculate MAs(n_t = sa0) and MAs(n_t = sa1). If conflicts occur during the MA computation, n_t can be replaced with the faulty value. If no conflict occurs, a node with different values in MAs(n_t = sa0) and MAs(n_t = sa1) can be selected as the substitute node n_s. Meanwhile, if there exist multiple nodes satisfying Condition 1, we select the node with the smallest size of MFFC as the n_s, which implicitly reduces the size and the depth of the circuit. Since the proposed NM technique is only executed during the mutation operation in the GA, which will be presented in Section IV, n_t is selected randomly.

We use an example to demonstrate the proposed NM technique in a DIG as shown in Fig. 1(b). Consider that v_3 is selected as n_t. Since the MA computation is dominator-based, only the dominators of n_t have to be considered, which are v_4 and v_5. The side-inputs of v_4 and v_5 are $(x, z) = (a, e)$ and $(y, z) = (c, v_1)$, respectively. According to different types of side-inputs, to compute MAs(v_3 = sa0), there are three combinations of value assignments to propagate the fault effect: $\{(a, e) = (1, 0), (c, v_1) = (0, 0)\}$, $\{(a, e) = (1, 0), (c, v_1) = (0, 1)\}$, and $\{(a, e) = (1, 0), (c, v_1) = (1, 1)\}$, which are listed in Tables II and III. Table II lists the steps of logic implication for each combination of values. The last column specifies the implication rules applied, shown in Fig. 5. For example, in the step (1) of the first combination $\{(a, e) = (1, 0), (c, v_1) = (0, 0)\}$, the assignment $(a, v_3) = (1, 1)$ implies $v_4 = 0$ using the implication rule *imp2*. Table III then summarizes the complete implied value assignments. As indicated in Table III, the assignments in the 1st row conflict such that they are discarded. Hence, MAs(v_3 = sa0) is obtained by the intersection of the assignments in the second row and third row, which are $\{v_3 = 1, a = 1, e = 0, \mathbf{v_1 = 1}, v_4 = 0, \mathbf{v_5 = 1}, v_2 = 0\}$. We then use the same procedure to get MAs(v_3 = sa1) = $\{v_3 = 0, a = 1, e = 0, c = 1, \mathbf{v_1 = 0}, v_4 = 0, \mathbf{v_5 = 0}\}$. As a result, v_1 and v_5 have different values between the two sets of MAs, which can be selected as the candidates of the substitute node. However, since replacing v_3 with v_5 will form a feedback loop in the circuit, we select v_1 rather than v_5 as the substitute node to replace the target node v_3.

NM is a strategy to ensure that the circuit size will shrink after the operation. Meanwhile, NM offers flexibility between efficiency and optimality. The complexity for computing the value of every node in MAs(n_t = sav) is as higher as performing a full simulation on the circuit. Even when considering only dominators, the computation for MAs still requires at most 3^n iterations when n_t has n dominators. This is because each dominator needs to consider up to three conditions to propagate the stuck-at fault. To elevate efficiency, we set a window for NM. This window covers nodes from l_{window} levels before n_t to l_{window} levels after n_t, and the total number of nodes within this window is less than n_{window}. When computing MAs, we only consider the nodes within this window. The dominators and n_s must also fall within this window. When there are many dominators within the window, we consider only one dominator at a time.

IV. ALS ON DIGS

A. Overall Algorithm

The overall algorithm of our GA-based ALS approach is shown in Algorithm 1. First, the circuit undergoes a node to constant phase. After performing node to constant, we evaluate the resulted circuit's ER. If the ER does not exceed the predefined ER constraint ε, we retain the modified circuit. Next, the circuit is encoded as a chromosome, followed by the generation of an initial population consisting of β_I individuals. Subsequently, k individuals are selected based on their fitness values.

After this phase, the iteration phase, as shown from Lines 8 to 26, begins. In this phase, crossover and mutation operations are performed. The crossover operation produces β_c individuals. In addition, for each parent, β_m individuals are generated through mutation. Consequently, $\beta_c + k \cdot \beta_m$ offspring are produced in one generation. Both the offspring and the parents are then evaluated to select the top k individuals, which will serve as the parents for the subsequent generation. Before proceeding to the next generation, the algorithm adjusts the next mutation level (ML) based on Algorithm 2 as shown in Line 20.

The proposed algorithm continues until the termination condition is met. In addition, we maintain the legal individual with the smallest size as shown from Lines 15 to 19 during each generation. In the final phase of our ALS, we perform

Algorithm 1 Overall Algorithm

Input $circuit$, p, ε, λ_m, β_I, β_c, β_m and k ;
$circuit_{ntc} \leftarrow$ NODE-TO-CONSTANT(circuit, p);
if $circuit_{ntc}$'s error $\leq \varepsilon$ **then**
  $circuit \leftarrow circuit_{ntc}$;
end if
$orig \leftarrow$ ENCODE the circuit ;
$candidates \leftarrow$ INITIALIZE β_I individuals with $orig$;
$parents \leftarrow$ SELECT k fitter individuals from $candidates$;
while TERMINATION CONDITION is not satisfied **do**
  $offspring \leftarrow$ CROSSOVER from $parents$ to produce β_c individuals;
  for $pr \in parents$ **do**
    $offspring \leftarrow$ MUTATE from pr to produce β_m individuals;
  end for
  append $parents$ to $offspring$;
  $parents \leftarrow$ SELECT k fitter individuals from $offspring$;
  for $pr \in parents$ **do**
    if pr's $ER \leq \varepsilon$ **and** pr's $AR \leq best$'s AR **then**
      $best \leftarrow pr$;
    end if
  end for
  SELF-ADJUSTED ML(λ_m, $offspring$);
end while
$circuit_{ntc} \leftarrow$ NODE-TO-CONSTANT(best, p);
if $circuit_{ntc}$'s error $\leq \varepsilon$ **then**
  $best \leftarrow circuit_{ntc}$;
end if
return $best$;

node to constant again to further reduce the size of the circuit and re-evaluate the error. If the resulting circuit meets the ER constraint ε, it is retained as the final best approximate circuit.

B. Node to Constant

In a circuit, there may be some nodes close to a constant value (0 or 1). Therefore, if we can identify these nodes and replace them with the constant values, the circuit size can be minimized without introducing many errors. Furthermore, since GA is a search algorithm, using this step can narrow the search space and increase the efficiency of GA.

We employ random simulation to determine whether a node approximates constant values. This involves applying a different set of random input patterns, X_{ntc}, to the original circuit and counting the number of values 0 and 1 in each node, denoted as |0| and |1|, respectively. If the probability of a node being 0 ((|0|/|X_{ntc}|)) is $\geq p$, we replace it with the constant 0. Conversely, if the probability of a node being 1 ((|1|/|X_{ntc}|)) is $\geq p$, we replace it with the constant 1. Here, p is a user-defined parameter.

C. Encoding Scheme

In GA, we encode each approximate circuit as a chromosome, where a chromosome is composed of multiple genes.

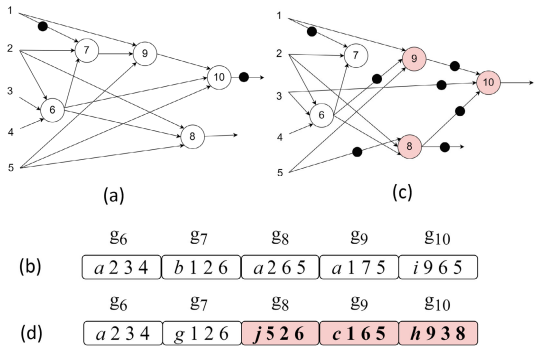

Fig. 7. Example of encoding a circuit as a chromosome and one of its individuals in the initial population. (a) Circuit represented in DIG. (b) Corresponding chromosome. (c) One of its generated individuals in the initial population is represented in DIG. (d) One of its generated individuals in the initial population is represented in a chromosome.

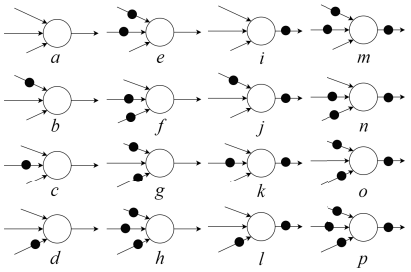

Fig. 8. 16 different gate types used in our encoding scheme.

First, we label each node in the circuit, including primary inputs (PIs) and constant value nodes, if any, in the topological ordering with a unique integer ID, as shown in the example of Fig. 7(a). The PIs and constant value nodes will be positioned at the forefront of the topological ordering, but inverters are excluded. Then, we define genes as Dot gates within the DIG. The encoding of the gene g_i with ID i is as shown as follows:

$$g_i = (t, u, v, m) \quad \max(u, v, m, |\text{PI}| + |\text{const}|) < i \tag{4}$$

where t is the gate type and u, v, and m are the IDs of the fanins of g_i. |PI| and |const| are numbers of PIs and constant value nodes in the circuit, respectively. The gate types are summarized in Fig. 8. All the gate types are three-input gates, including input-negated or output-negated variants of Dot gates. Fig. 7(b) illustrates the chromosome corresponding to the circuit in Fig. 7(a). Taking g_7 from Fig. 7(b) as an example, denoted as $(b, 1, 2, 6)$, where b represents input-x-negated version of Dot gate, and 1, 2, and 6 represent that the inputs x, y, and z of the Dot gate are connected to the PI with ID 1, the PI with ID 2, and g_6, respectively.

Loop prevention can be achieved through topological order numbering for IDs and our encoding scheme with (4). The encoding scheme ensures a valid combinational structure and allows for the change of the function in the gene's location by varying the gate type and input permutation. This scheme facilitates the exploration of functions within the NPN equivalence class [8], [15] of the Dot function. Most importantly, this encoding scheme is a one-to-one correspondence between chromosomes and approximate circuits, meaning that a chromosome can be decoded back to a unique circuit structure.

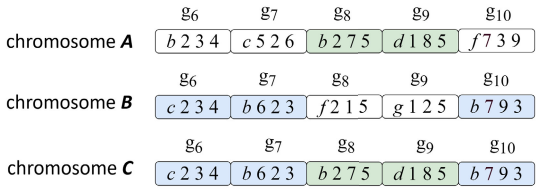

Fig. 9. Example of the crossover operation on two chromosomes, where chromosome A and chromosome B are parents and chromosome C is the generated offspring.

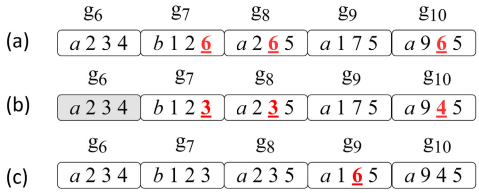

Fig. 10. Two examples of the mutation operations. (a) Original chromosome. (b) Mutated chromosome with respect to (a). (c) Mutated chromosome with respect to (b).

D. Initial Population

Since GA requires generating a set of diverse individuals initially, which is called the initial population, we can modify the chromosome of the original circuit to create such a set. The process of generating one individual is conducted as follows: we randomly select a gene position i and split the chromosome into two parts at i. We then randomly select one part and generate a new set of genes satisfying (4) to replace all the genes in this part of the chromosome. For example, in Fig. 7(c) and (d), suppose $i = 8$, the chromosome is divided into two parts: $[g_6, g_7, g_8]$ and $[g_8, g_9, g_{10}]$. Suppose the latter part is selected, all the genes in this part are replaced by randomly generated genes, $(j, 5, 2, 6)$, $(c, 1, 6, 5)$, and $(h, 9, 3, 8)$, respectively.

With this setup, some genes may become inactive. A gene within a chromosome is active if, when decoded back to a circuit, there is a path from the gate corresponding to the active gene to at least one PO. Conversely, if a gene is inactive, there is no path from the gate corresponding to the gene to any PO. Inactive nodes represent the absence of the nodes in the corresponding positions, meaning the number of active genes in a chromosome exactly represents the area (node count) of the individual circuit. By introducing perturbations to the chromosome, we can generate a set of individuals with varying degrees of structural differences and diverse node counts from the original circuit. Note that to determine the area of the circuit represented by a chromosome, we can apply a depth-first search from the POs to assess the number of active nodes for each chromosome.

E. Crossover and Mutation

Our crossover operation proceeds as follows: for each chromosome A, we randomly pair it with another chromosome B as parents to generate a new offspring represented by chromosome C. For each g_i in chromosome C, we randomly choose one gene either from chromosome A or B in the same gene position i. In the example shown in Fig. 9, the offspring chromosome C inherits g_8 and g_9 from A and g_6, g_7, and g_{10} from B.

Our mutation operation is as follows: we randomly select λ_m active genes within a chromosome as the targets for mutation. For each target gene, we will either 1) turn it inactive by rewiring all of its fanouts to any other node; 2) randomly rewire one of its fanins; or 3) randomly change its gate type. We explain these actions using the example in Fig. 10. Assume that we would like to turn g_6 of this chromosome inactive by action (1). We rewire each fan-out of g_6, which are g_7, g_8, and g_{10}, to any other node, as shown in Fig. 10(b). On the other hand, an inactive gene may be turned active again by action (2). As shown in Fig. 10(b), g_6 is initially inactive. If we select g_9 to be mutated, there is a possibility of turning g_6 active again by rewiring one of g_9's fanins to g_6, as shown in Fig. 10(c).

Unlike other greedy ALS methods, allowing the circuit size to increase in our algorithm—specifically, permitting an inactive node to become active again—may help the algorithm avoid getting stuck in a local optimum.

F. Fitness Function

Each chromosome's fitness value is calculated based on a fitness function for assessing the quality of the individual. The individuals with higher fitness values will be selected to be the parents in the next generations to carry out crossover and mutation operations. The fitness function in our approach is defined as follows:

$$\text{Fitness}_1 = \frac{1}{W_E \cdot g(\text{ER}) \cdot (\text{ER} + \text{ARHD}) + W_A \cdot \text{AR}} \tag{5}$$

where AR is the area ratio between the approximate circuit and the original circuit. W_E and W_A are the weights for error evaluation and area evaluation, respectively. $g(\text{ER})$ is a penalty function with ER as its variable.

The weights W_E and W_A are used to provide the configurability to adjust whether the algorithm should focus more on reducing the area or minimizing the error. The ARHD is incorporated into the fitness function to differentiate individuals with distinct numbers of erroneous POs. This idea aims to enhance the precision of the fitness value. Since we prefer to retain circuits with fewer erroneous POs for offspring generation, as this implies fewer error sites in the circuit, we use ARHD as a secondary error metric. Instead of using the Hamming distance directly, ARHD ensures that the influence of the primary metric, ER, is not overshadowed. By incorporate ARHD into the fitness function, individuals with higher ARHD will result in a lower fitness value, thereby decreasing their chances of survival. $g(\text{ER})$ is used to penalize the individuals with an ER exceeding the constraint. Its function is defined as

$$g(\text{ER}) = \begin{cases} 1, & \text{if ER} \le \varepsilon \\ e^{\frac{\text{ER}}{\varepsilon}}, & \text{if ER} > \varepsilon \end{cases} \tag{6}$$

where ε is the ER constraint. An exponential function, $e^{(\text{ER}/\varepsilon)}$, is used to impose a significantly large penalty when the ER exceeds ε.

Although our approach mainly focuses on logic synthesis for reducing the area of DIG circuits, we can extend the fitness

function to incorporate the depth of the circuits as well. Our fitness function can be modified to include the depth of the circuits as follows:

$$\text{Fitness}_2 = \frac{1}{\frac{1}{\text{Fitness}_1} + W_D \cdot \text{DR}} \tag{7}$$

where DR is the depth ratio between the approximate circuit and the original one. W_D is the weight for the depth ratio in the fitness function.

Furthermore, when NMED is used as the error evaluation metric, the fitness function and the penalty function in our approach are defined as follows:

$$\text{Fitness}_3 = \frac{1}{W_E \cdot g(\text{NMED}) \cdot \text{NMED} + W_A \cdot \text{AR}} \tag{8}$$

$$g(\text{NMED}) = \begin{cases} 1, & \text{if NMED} \leq \varepsilon \\ e^{\frac{\text{NMED}}{\varepsilon}}, & \text{if NMED} > \varepsilon \end{cases} \tag{9}$$

where ε represents the NMED constraint. Similar to the ER-based fitness function, the penalty function increases exponentially as NMED exceeds the constraint.

G. NM-Enhanced Mutation

In GA, the removal of gates is in a random manner, causing the algorithm to explore an unknown search space. Because GA operations can change the circuit structure and may generate more different do not cares, NM may utilize these do not cares to shrink the circuit size without affecting the circuit's functionality. Unlike other optimization methods, NM can maintain the topological ordering of the circuit. This means that individuals modified by NM can be iterated in the subsequent generations within GA because the encoding style remains consistent with the original one. Therefore, we develop an NM-enhanced mutation (NM-M) to increase the effectiveness of our GA.

In the action (1) of mutation, if the chromosome is legal, i.e., an individual's ER is lower than the ER constraint, we first use NM to assist the mutation. This means that we treat the selected gene as n_t and compute its MA. If n_s is successfully found, the mutation is finished, and the ER of the chromosome remains unchanged, thus ensuring the chromosome is legal. If n_s cannot be found, the random mutation method will be applied.

Although the process of NM consumes more time as compared with random mutation, it can reduce the overall time GA spends. As illustrated in Fig. 3(b), NM advances the solution from point A to point S: reducing the area but not increasing the ER, which pushes the solution closer to the Pareto front.

H. Self-Adjusted ML

To improve the scalability of our algorithm, we allow the algorithm to self-adjust its ML based on the quality of the population in each generation. Specifically, if the population in the current generation has a lower average ER, we will permit more genes, indicated by λ_m, to be mutated during the mutation operation. Conversely, if the population's average ER in the current generation is higher, the number of genes subjected to mutation will be decreased.

Algorithm 2 Self-Adjusted ML

```
if AvgER < AvgER_low then
    λ_m = (λ_m + 1) × 1.3;
else if AvgER > AvgER_high then
    λ_m = (λ_m − 1) × 0.7;
end if
```

The algorithm of self-adjusted ML is shown in Algorithm 2, where AvgER denotes the average ER of the mutated individuals in the current generation. $\text{AvgER}_{\text{low}}$ and $\text{AvgER}_{\text{high}}$ denote user-defined thresholds for determining when AvgER is low or high, respectively.

V. Experimental Results

We implemented the proposed approach using C++ [3]. The platform is an Intel Core i7-11700 2.5 GHz running Ubuntu 22.04 LTS with 32-GB memory. We adopted circuits that have been converted into DIG from the EPFL 2015 [5] in [29] as our benchmarks [1]. In [29], to demonstrate the high expressive ability of DIG, they alternatively performed 4-LUT mapping and exact node resynthesis on the benchmarks [5] until the circuit size could no longer be reduced. Therefore, we consider these benchmarks to be well-optimized circuits.

Unless otherwise specified, the parameters of the experiments were set as follows: the number of patterns for error evaluation $|X| = 122\,880$, the number of sampling patterns in node to constant phase $|X_{\text{ntc}}| = 122\,880$, the probability threshold in node to constant phase $p = 100\% - \varepsilon \times 0.1$, ER constraint $\varepsilon = 5\%$. The parameters for GA were set as follows: the initial value of $\lambda_m = 1$, $\beta_I = 500$, $\beta_c = 300$, $\beta_m = 2$, and $k = 300$. The thresholds for ML self-adjustment were as follows: $\text{AvgER}_{\text{low}} = \varepsilon$ and $\text{AvgER}_{\text{high}} = \varepsilon \times 2$. The parameters for the window of NM were set as follows: $l_{\text{window}} = 5$ and $n_{\text{window}} = 300$. The weights in the fitness function were as follows: $W_E = 1$ and $W_A = 100$. The termination condition for the GA was set as follows: the algorithm terminates when the best chromosome for each generation remains unchanged for 50 consecutive generations, or when the 3600-s time limit is met.

A. Effectiveness of the Proposed Approach

In the first experiment, we compared our ALS approach with the one proposed by Su et al. in [42]. Su et al. [42] can be considered as the state-of-the-art in ALS. In [42], they estimated the error by applying different approximate local transformations (ALTs), which include removing literals, removing nodes, etc., in each iteration. They used simulation and logic implication to determine the impact of changes, and record this information in a large matrix. Based on this matrix, they greedily select the best ALT to apply to the circuit iteratively. The iteration continues until applying the best ALT on the circuit cannot meet the given ER constraint.

The experimental results are shown in Table IV. Columns 1–4 list the information of the benchmarks, including their name, the number of PIs and POs, depth, and size. Columns 5–9 list the results of our approach. Columns 10–14 list the

TABLE IV
EXPERIMENTAL RESULTS OF PROPOSED ALGORITHM COMPARED WITH [42] UNDER $\varepsilon = 5\%$ AND TIME LIMIT = 3600 S

Benchmarks				Ours					[42]				
Circuit	|PI|/|PO|	Depth	|Dot|	Depth	|Dot|	Reduc.(%)	ER(%)	Time(s)	Depth	|Dot|	Reduc.(%)	ER(%)	Time(s)
ctrl	7/26	8	117	7	80	31.62	4.59	19.50	6	83	29.06	3.11	10.58
int2float	11/7	19	202	13	120	40.59	4.60	5.85	12	139	31.19	4.60	19.46
router	60/30	32	202	2	3	98.51	0.60	48.12	2	3	98.51	0.59	45.82
dec	8/256	3	304	3	297	2.30	0.00	50.12	3	292	3.95	4.69	353.84
adder	256/129	255	511	255	510	0.20	0.00	80.18	255	511	0.00	0.00	510.95
cavlc	10/11	**19**	644	**21**	539	16.30	4.95	109.56	20	535	16.93	4.99	20.96
priority	128/8	125	664	3	8	98.80	4.41	197.85	4	10	98.49	2.72	247.78
i2c	147/142	18	1,105	13	601	45.61	4.94	62.15	9	452	59.10	4.68	765.05
max	512/130	229	2,230	33	820	63.23	4.98	120.85	64	1,198	46.28	0.01	***3,600.00***
bar	135/128	**11**	2,246	**14**	1,978	11.93	0.00	34.72	14	1,977	11.98	0.01	2,980.56
sin	24/25	149	3,995	122	3,143	21.33	4.70	143.85	129	3,556	10.99	4.55	1,743.87
voter	1,001/1	64	7,385	45	5,210	29.45	0.24	237.95	41	5,125	30.60	0.17	2,460.43
arbiter	256/129	63	8,515	21	1,719	79.81	4.38	616.94	63	8,214	3.53	0.01	***3,600.00***
square	64/128	251	12,743	251	12,741	0.02	3.34	1,607.11	251	12,743	0.00	0.00	***3,600.00***
sqrt	128/64	6,045	17,027	4,356	12,172	28.51	4.84	2,680.85	4,216	12,398	27.19	0.01	***3,600.00***
multiplier	128/128	261	17,273	261	17,273	0.00	0.00	394.46	263	17,227	0.27	0.00	***3,600.00***
log2	32/32	318	23,144	295	19,434	16.03	4.85	734.48	318	23,089	0.24	0.01	***3,600.00***
mem_ctrl	1,204/1,231	130	37,114	48	16,456	55.66	4.78	*3,600.00*	60	21,547	41.94	0.01	***3,600.00***
div	128/128	4,401	43,421	893	5,509	87.31	4.80	*3,600.00*	3,097	14,832	65.84	0.01	***3,600.00***
Average	-	**652.68**	9412.74	**350.32**	5190.16	38.28	3.11	754.98	**464.58**	6522.68	30.32	1.59	1,997.86

TABLE V
COMPARISON OF EXPERIMENTAL RESULTS OF [42] UNDER $\varepsilon = 5\%$ AND TIME LIMIT = 7200 S, AND OURS UNDER TIME LIMIT = 3600 S

Circuit	|Dot|	Reduc. (%)	ER (%)	Time (s)	ΔReduc. 3600s	ΔReduc. Ours
max	802	64.04	4.40	5,464.60	17.76%	0.81%
arbiter	7,828	8.07	0.01	*7,200.00*	4.53%	-71.74%
square	12,743	0.00	0.00	*7,200.00*	0.00%	-0.02%
sqrt	12,398	27.19	0.01	*7,200.00*	0.00%	-1.33%
multiplier	17,183	0.52	0.00	*7,200.00*	0.25%	0.52%
log2	23,072	0.31	0.01	*7,200.00*	0.07%	-15.72%
mem_ctrl	21,332	42.52	0.01	*7,200.00*	0.58%	-13.14%
div	14,832	65.84	0.01	*7,200.00*	0.00%	-21.47%

results of [42]. The program from [42] is publicly available at [2] and can accept DIG circuits as input. However, since some benchmarks caused the program to timeout without producing results, we modified the code to ensure it reports the best solution achieved at the time of the timeout. Subsequently, the output is mapped into a DIG and optimized using the method in [29].

According to the experimental results, our approach achieved an average reduction of 38.28% node count, which is higher than 30.32% achieved by [42]. As for the average CPU time, our approach required less than half of theirs, with 754.98 s compared to 1997.86 s. For some arithmetic benchmarks, such as *adder*, *square*, and *multiplier*, both approaches cannot reduce the node count. In addition, we can see that for larger circuits, such as *max* and those from *arbiter* to *div*, their method failed to progress after it reached an ER of 0.01%. This was possibly due to their greedy strategy, which caused the algorithm to get stuck in a local optimum. On the other hand, instead of continuously reducing the circuit size, our approach allows perturbations to escape from the local optimal points. For the depth of the circuits, our approach reduces the average circuit depth from 652.68 to 350.32, whereas [42] achieves only 464.58. Although depth was not considered in this experiment of our approach, we observe that higher area reductions may usually lead to higher depth reductions. However, it is not the case for the benchmarks *cavlc* and *bar*. Their areas were reduced, but the depth was increased. The reason is probably because our approach produced solutions along the Pareto front and prioritized the one with the lower area.

Considering that the benchmarks exceeding the time limit might need more CPU time, we conducted the second experiment, where we extended the time limit to 7200 s, and conducted their method again. The experimental results are shown in Table V. Columns 2–5 list the results of resultant circuits. Column 6 lists the difference in reduction ratio compared to the results under the original 3600-s time limit. Column 7 lists the difference in reduction ratio compared to our results under the 3600-s time limit. According to Table V, we can see that only *max* and *multiplier* successfully produced smaller circuits, less than 1%, under the 7200-s time limit as compared to our approach. However, for the other benchmarks, their results were still inferior to those achieved by our approach.

To further validate the effectiveness of the proposed GA, we conducted the third experiment by setting $\varepsilon = 10\%$. The results are shown in Table VI. The experimental results show that with a more relaxed ER, smaller sizes of resultant circuits can be achieved in most cases. Moreover, as compared to [42], our approach performed better, achieving a higher average reduction ratio of 41.35% compared to 33.44%, as well as a lower average CPU time of 1011.32 s compared to 2026.42 s. It is worth noting that for *router*, both approaches produced circuits that only consist of constant value nodes, resulting in zero circuit depth and zero circuit size. However, due to the difference in constant values of the resultant circuits, the ER differ slightly with values of 7.02% and 6.81%, respectively.

To include circuit depth into our consideration, we conducted the fifth experiment. We used the fitness function shown as (7) in our approach. The weights in the fitness function were as follows: $W_E = 1$, $W_A = 50$, and $W_D = 50$. The ER constraint ε was set to 5%. The experimental results are shown in Table VII. We compared these results with those from our previous experiments shown in Table IV. Columns 2 and 3 list the results of resultant circuits. Column 4 lists

TABLE VI
EXPERIMENTAL RESULTS OF PROPOSED ALGORITHM COMPARED WITH [42] UNDER $\varepsilon = 10\%$ AND TIME LIMIT = 3600 S

Benchmarks				Ours					[42]				
Circuit	|PI|/|PO|	Depth	|Dot|	Depth	|Dot|	Reduc.(%)	ER(%)	Time(s)	Depth	|Dot|	Reduc.(%)	ER(%)	Time(s)
ctrl	7/26	8	117	6	77	34.19	9.43	41.96	6	84	28.21	9.30	6.15
int2float	11/7	19	202	11	69	65.84	9.85	32.77	10	93	53.96	8.29	13.95
router	60/30	32	202	**0**	**0**	100.00	**7.02**	0.03	**0**	**0**	100.00	**6.81**	360.31
dec	8/256	3	304	3	287	5.59	6.19	25.27	3	279	8.22	9.82	352.56
adder	256/129	255	511	255	510	0.20	0.00	10.28	254	510	0.20	6.23	536.84
cavlc	10/11	19	644	19	523	18.79	9.94	34.29	21	416	35.40	9.84	36.09
priority	128/8	125	664	3	6	99.10	6.19	277.98	2	3	99.55	6.17	247.91
i2c	147/142	18	1,105	14	452	59.10	8.84	486.30	9	360	67.42	9.61	857.25
max	512/130	229	2,230	22	799	64.17	8.60	269.90	65	1,184	46.91	0.07	*3,600.00*
bar	135/128	11	2,246	14	1,978	11.93	0.00	28.37	15	1,976	12.02	6.27	3,066.44
sin	24/25	149	3,995	122	3,149	21.18	9.84	606.74	127	3,490	12.64	9.96	1,689.58
voter	1,001/1	64	7,385	45	5,158	30.16	9.33	390.84	43	5,138	30.43	7.08	2,534.81
arbiter	256/129	63	8,515	21	1,680	80.27	9.90	1,052.53	63	8,223	3.43	0.04	*3,600.00*
square	64/128	251	12,743	251	12,740	0.02	9.83	1,201.01	251	12,743	0.00	0.00	*3,600.00*
sqrt	128/64	6,045	17,027	4,433	12,155	28.61	9.89	*3,600.00*	4,211	12,137	28.72	0.01	*3,600.00*
multiplier	128/128	261	17,273	261	17,272	0.01	9.47	650.46	263	17,227	0.27	0.00	*3,600.00*
log2	32/32	318	23,144	295	19,061	17.64	9.82	3,306.44	318	23,108	0.16	0.01	*3,600.00*
mem_ctrl	1,204/1,231	130	37,114	53	16,267	56.17	8.34	*3,600.00*	60	21,547	41.94	0.01	*3,600.00*
div	128/128	4,401	43,421	661	3,210	92.61	7.40	*3,600.00*	3,095	14,843	65.82	0.01	*3,600.00*
Average	-	652.68	9412.74	341.53	5020.68	**41.35**	7.89	**1011.32**	464.00	6492.68	**33.44**	4.71	**2026.42**

TABLE VII
EXPERIMENTAL RESULTS OF THE PROPOSED ALGORITHM WITH CIRCUIT DEPTH CONSIDERED IN THE FITNESS FUNCTION UNDER $\varepsilon = 5\%$ AND TIME LIMIT = 3600 S

Circuit	Depth	|Dot|	DR(%)	DR(%) Table IV	ΔDR(%)
ctrl	8	89	100.00%	87.50%	12.50%
int2float	13	132	68.42%	68.42%	0.00%
router	3	4	9.38%	6.25%	3.13%
dec	3	304	100.00%	100.00%	0.00%
adder	255	511	100.00%	100.00%	0.00%
cavlc	17	581	89.47%	110.53%	-21.05%
priority	3	7	2.40%	2.40%	0.00%
i2c	13	777	72.22%	72.22%	0.00%
max	28	875	12.23%	14.41%	-2.18%
bar	14	1978	127.27%	127.27%	0.00%
sin	132	3247	88.59%	81.88%	6.71%
voter	41	5245	64.06%	70.31%	-6.25%
arbiter	17	1604	26.98%	33.33%	-6.35%
square	251	12743	100.00%	100.00%	0.00%
sqrt	4356	12172	72.06%	72.06%	0.00%
multiplier	261	17273	100.00%	100.00%	0.00%
log2	277	20338	87.11%	92.77%	-5.66%
mem_ctrl	46	17598	35.38%	36.92%	-1.54%
div	569	9981	12.93%	20.29%	-7.36%
Average	**331.95**	**5550.47**	-	-	-1.48%

the depth ratio of the resultant circuits to the original ones. Column 5 lists the depth ratio using the depth values of the resultant circuits under our approach in Table IV. Column 6 lists the difference between the values in Columns 4 and 5. According to the experimental results, the average circuit depth decreased from 350.32 to 331.95, whereas the average circuit size increased from 5190.16 to 5550.47. We believe that this is because the algorithm considers depth optimization rather than solely minimizing circuit size.

B. Effectiveness of NM-M and GA Elements

Our approach integrates several elements, such as NM-M, ARHD, and self-adjusted ML. To recognize their contributions to our approach, we conducted an ablation experiment, where each element was removed individually while keeping the others unchanged. The experimental results are shown in Fig. 11. The left vertical axis indicates the average node count ratio, which is defined as the ratio of the reduced circuit size to the original circuit size, averaged across all benchmarks. The right vertical axis indicates the average time ratio, which is the ratio of CPU time compared to the complete approach, averaged across all benchmarks.

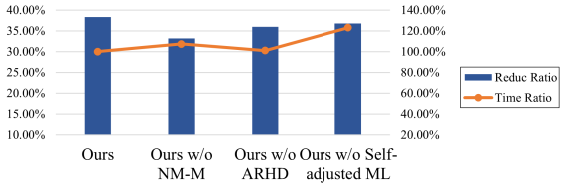

Fig. 11. Average node count reduction rates and average time ratios from the ablation experiments.

According to Fig. 11, we can see that NM-M leads to better outcomes. The average node reduction ratio increased from 33% in the approach without NM-M to 38% in the complete approach, while the average time ratio decreased from 107% to 100%, respectively. This indicates that although NM-M requires additional computation, it assists GA in finding better solutions more quickly within the given constraints and offers better overall performance. For the experimental results of removing ARHD, the reduction ratio decreased by 3% to 35%. This indicates that ARHD is effective for the selection of individuals. As for the experimental results of removing the self-adjusted ML, the CPU time increased to 123% compared to one of the complete approach, while the node reduction ratio decreased from 38% to 36%. This is because the ML was dynamically adjusted across different circuit sizes and could adapt by observing the mutation results in each iteration, allowing the approach to find the optimal solution more efficiently.

C. Statistical Evaluation of ALS Results

Since our approach involves randomness, we performed multiple runs of our approach and report the mean and standard deviation for the node count and depth. The number

TABLE VIII
MEAN AND STANDARD DEVIATION OF EXPERIMENTAL RESULTS OF THE PROPOSED ALGORITHM OVER MULTIPLE RUNS UNDER $\varepsilon = 5\%$ AND TIME LIMIT = 3600 S

Circuit	Depth			|Dot|		
	μ	σ	$P_{\mu\pm\sigma}$ (%)	μ	σ	$P_{\mu\pm\sigma}$ (%)
ctrl	7.31	1.01	75.00	85.94	3.13	68.75
int2float	14.06	1.06	87.50	111.69	10.33	**62.50**
router	2.38	0.50	**62.50**	3.38	0.50	**62.50**
dec	3.00	0.00	100.00	303.56	1.75	93.75
adder	255.00	0.00	100.00	510.94	0.25	93.75
cavlc	18.31	0.79	93.75	541.75	9.53	**56.25**
priority	3.00	0.73	68.75	7.06	0.77	**62.50**
i2c	12.13	1.36	87.50	526.19	41.24	68.75
max	29.81	4.65	**62.50**	827.13	10.32	**62.50**
bar	14.94	0.25	93.75	1950.56	7.38	93.75
sin	122.38	1.50	93.75	3180.25	36.80	75.00
voter	45.94	2.05	81.25	5135.56	20.22	93.75
arbiter	18.56	4.13	**62.50**	1568.88	113.23	75.00
square	251.00	0.00	100.00	12742.06	0.44	81.25
sqrt	3247.30	766.53	70.00	8672.40	2424.79	70.00
multiplier	261.00	0.00	100.00	17273.00	0.00	100.00
log2	300.50	3.81	81.25	19995.13	371.63	68.75
mem_ctrl	47.50	1.91	78.57	14985.79	930.59	**64.29**
div	788.13	82.59	68.75	4975.75	461.73	75.00
Average	-	-	75.16	-	-	82.49

of samples we used was 30, which means that we performed the approach for 30 times. The results are shown in Table VIII. Columns 2 and 3 list the mean value (μ) and standard deviation value (σ) of circuit depth, respectively. Column 4 lists the $P_{\mu\pm\sigma}$ value, which represents the percentage of the samples that fall within one standard deviation from the mean, i.e., $[\mu - \sigma, \mu + \sigma]$. Columns 5–7 list the corresponding values for the node count. Based on the 68–95–99.7 rule, if the results follow a normal distribution, the $P_{\mu\pm\sigma}$ value should be greater than 68% approximately. According to the experimental results, we observed that for more than half of the benchmarks, the $P_{\mu\pm\sigma}$ values for both depth and node count exceed 68%, indicating that the approach produces stable outcomes for these benchmarks. However, for some benchmarks, the $P_{\mu\pm\sigma}$ values for either depth or node count fall below 68%, indicating higher variability. On average, the $P_{\mu\pm\sigma}$ values for both depth (75.16%) and node count (82.49%) exceed the expected 68%.

D. Effectiveness of the Proposed Approach Under NMED Constraint

We also conducted the experiments where NMED is the error constraint. As NMED is commonly used for arithmetic circuits, we conducted experiments only on the arithmetic circuits from [1]. Since the program from [42] is not applicable to DIG circuits under NMED constraints, we modified the experimental setup. In [42], the results were reported after technology mapping using the MCNC Library [53]. Therefore, although Dot gates are typically expected to be used in VNWFET technology, for comparison purposes, we also mapped our results using the MCNC Library in the same manner. The results are shown in Table IX. We conducted experiments under two NMED constraints: 0.2% and 0.02%. Columns 2 and 3 list the area and delay of the original benchmarks mapped using the MCNC Library. Columns 4–6 list the results of our approach under the NMED constraint of 0.2%. Columns 7–9 list the results of the method proposed in [42]. Columns 10–15 list the corresponding results under the NMED constraint of 0.02%. The area ratio and delay ratio refer to the ratios of the area and delay of the mapped circuits to the original circuit's area and delay, respectively. In addition, it is worth mentioning that for the *div* benchmark, the output numerical value is defined as the summation of the quotient and remainder. According to the results, for all the EPFL benchmarks, the method in [42] failed to produce a circuit within the given runtime limit, as it either terminated without producing better results or timed out without generating any output. This suggests that the EPFL benchmarks are too large to be efficiently handled by the method in [42] under the NMED constraints. To further assess the comparison, we additionally selected three small arithmetic circuits from [42] with area falling into three ranges: less than 1000, between 1000 and 2000, and greater than 2000. These benchmarks are *sad*, *ksa32*, and *eudist*, as shown in the last three benchmarks of Table IX. The final row of the table reports the average area ratio and average CPU time across these three benchmarks. Under the 0.2% NMED constraint, our approach yielded an average area ratio of 67.47% compared to 66.49% for the method in [42]. However, our approach required significantly less CPU time, 434.09 s on average, as compared to the method in [42], 3249.15 s on average. Under the stricter 0.02% constraint, our approach achieved better average area ratio (74.27% versus 77.64%) while still using much less CPU time, 886.96 s on average, compared to 2667.69 s for [42].

E. Comparison With the ALS Method for Selection of Multiple Changes Per Iteration

In the last experiment, we compare our approach with the method proposed in [49], where multiple approximate changes are applied to the circuit within a single iteration, and strategies are introduced for selecting these changes to accelerate the overall algorithm. In the experiments in [49], they selected several arithmetic benchmarks from the EPFL benchmarks in AIG format and performed their method under an ER constraint of 0.1%. They reported their results in terms of area and delay, both normalized to the area and delay of the INV_X1 gate in the MCNC Library [53]. Since our original approach appeared aggressively under such a tight error constraint, 0.1%, we adjusted our approach by gradually relaxing the error constraint, i.e., executing our method sequentially with error constraints of 0.02%, 0.05%, 0.08%, and 0.1%, in this experiment. The experimental results are shown in Table X. Columns 2 and 3 list the area and delay of the benchmarks mapped using the MCNC Library. Columns 4–6 list the results of our approach, while Columns 7–9 list the results of the method proposed in [49]. It is worth to note that the data in Columns 2, 3, and 7–9 are identical to those reported in [49]. The experimental results show that, on average, our approach achieves approximately 7% improvement in area and 5% improvement in delay. As mentioned in Section I, the method in [49] aimed to identify a set of conflict-free independent approximate changes per iteration and applied some of them in a greedy manner to reduce circuit size until no

TABLE IX
EXPERIMENTAL RESULTS OF THE PROPOSED ALGORITHM UNDER NMED CONSTRAINT AND TIME LIMIT = 3600 S

Benchmarks			NMED constraint = 0.2%						NMED constraint = 0.02%					
			Ours			[42]			Ours			[42]		
Circuit	Area	Delay	Area Ratio(%)	Delay Ratio(%)	Time(s)	Area Ratio(%)	Delay Ratio(%)	Time(s)	Area Ratio(%)	Delay Ratio(%)	Time(s)	Area Ratio(%)	Delay Ratio(%)	Time(s)
max	4,448	485.4	25.31	6.96	1,110.17	**100.00**	100.00	33.70	27.90	6.86	2,704.15	**100.00**	100.00	32.32
square	37,927	400.8	99.99	56.64	*3,600.00*	**100.00**	100.00	34.94	99.94	56.64	*3,600.00*	**100.00**	100.00	36.98
sqrt	44,512	3,568.1	2.31	2.57	348.65	**100.00**	100.00	660.05	3.43	4.96	745.19	**100.00**	100.00	665.80
div	47,081	3,420.0	30.02	31.16	*3,600.00*	**100.00**	100.00	*3,600.00*	41.26	39.56	*3,600.00*	**100.00**	100.00	*3,600.00*
multiplier	56,653	427.8	82.45	104.02	*3,600.00*	**100.00**	100.00	95.67	82.80	104.58	*3,600.00*	**100.00**	100.00	95.22
sad	999	70.9	79.58	51.64	295.01	55.46	117.80	2,568.50	82.88	91.82	281.49	83.98	58.39	803.07
ksa32	1,128	27.0	42.20	77.72	332.65	48.58	62.22	3,585.14	56.29	89.26	424.04	51.68	61.11	*3,600.00*
eudist	2,731	87.3	80.63	62.22	674.61	95.42	61.63	3,593.80	83.63	97.82	1,955.34	97.25	61.63	*3,600.00*
Average	-	-	**67.47**	-	**434.09**	**66.49**	-	**3249.15**	**74.27**	-	**886.96**	**77.64**	-	**2667.69**

TABLE X
EXPERIMENTAL RESULTS OF THE PROPOSED ALGORITHM COMPARED WITH [49] UNDER $\varepsilon = 0.1\%$

Benchmarks			Ours			[49]		
Circuit	Area	Delay	Area Ratio (%)	Delay Ratio (%)	Time(s)	Area Ratio (%)	Delay Ratio (%)	Time (s)
div	47,081	3,420.0	35.78	39.94	7,185	26.93	27.22	8,703
log2	64,914	541.2	76.63	76.87	4,217	90.60	94.38	4,535
sin	12,169	254.8	76.63	76.92	832	96.58	96.70	589
sqrt	44,512	3,568.1	74.63	100.98	5,684	78.30	79.56	5,556
square	37,927	400.8	92.20	73.60	893	99.78	98.95	617
Average	-	-	**71.17**	**73.66**	3,762	**78.44**	**79.36**	4,000

further reduction is possible. In contrast, our approach allows arbitrary changes to the circuit and provides opportunities for recovering deleted nodes, enabling a nongreedy strategy. These are critical elements enabling our approach to achieve better results.

VI. CONCLUSION AND FUTURE WORK

In this article, we propose an ALS approach–NM-enhanced GA-based ALS, with an emphasis on reducing the size of DIG circuits while maintaining acceptable ERs. By integrating NM with GA, we demonstrate a significant improvement of our approach on ALS. Our future work is to enhance the effectiveness of NM and to integrate don't-cares-aware GA operations into our approach.

REFERENCES

[1] *Three-input-gates-TCAD2020*. Accessed: Feb. 2, 2024. [Online]. Available: https://github.com/mdsudara/three-input-gates-tcad2020

[2] *Vecbee*. Accessed: Feb. 2, 2024. [Online]. Available: https://github.com/sjtu-ectl/vecbee

[3] *Dig-approx*. Accessed: Feb. 15, 2025. [Online]. Available: https://github.com/yitingl/dig-approx

[4] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Hoboken, NJ, USA: Wiley, 1989.

[5] L. Amarú, P. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, vol. 256, no. 128, 2015, Art. no. 214335.

[6] M. Barbareschi, S. Barone, N. Mazzocca, and A. Moriconi, "A catalog-based AIG-rewriting approach to the design of approximate components," *IEEE Trans. Emerg. Topics Comput.*, vol. 11, no. 1, pp. 70–81, Jan. 2023.

[7] A. Bosio et al., "Emerging technologies: Challenges and opportunities for logic synthesis," in *Proc. 24th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2021, pp. 93–98.

[8] C. H. Chang, "Spectral techniques in digital logic," Ph.D. dissertation, Nanyang Technological University, Singapore, 1997.

[9] Y.-C. Chen and C.-Y. Wang, "Fast detection of node mergers using logic implications," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design - Dig. Tech. Papers*, Nov. 2009, pp. 785–788.

[10] Y.-C. Chen and C.-Y. Wang, "Fast node merging with don't cares using logic implications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 11, pp. 1827–1832, Nov. 2010.

[11] K. De Jong, "Learning with genetic algorithms: An overview," *Mach. Learn.*, vol. 3, nos. 2–3, pp. 121–138, Oct. 1988.

[12] J. Echavarria, S. Wildermann, O. Keszöcze, and J. Teich, "Probabilistic error propagation through approximated Boolean networks," in *Proc. DAC*, Feb. 2020, pp. 1–6.

[13] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Cham, Switzerland: Springer, 2015.

[14] S. Froehlich, D. Große, and R. Drechsler, "Error bounded exact BDD minimization in approximate computing," in *Proc. IEEE 47th Int. Symp. Multiple-Valued Log. (ISMVL)*, May 2017, pp. 254–259.

[15] E. Goto and H. Takahasi, "Some theorems useful in threshold logic for enumerating Boolean functions," in *Proc. IFIP Congr.*, 1962, pp. 747–752.

[16] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[17] R. Hrbacek, V. Mrazek, and Z. Vasicek, "Automatic design of approximate circuits by means of multi-objective evolutionary algorithms," in *Proc. Int. Conf. Design Technol. Integr. Syst. Nanosc. Era (DTIS)*, Apr. 2016, pp. 1–6.

[18] S. Karakatic, V. Podgorelec, and M. Hericko, "Optimization of combinational logic circuits with genetic programming," *Electron. Electr. Eng.*, vol. 19, no. 7, pp. 86–89, Sep. 2013.

[19] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2013, pp. 130–137.

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[21] C.-C. Ko, C.-C. Lin, Y.-C. Chen, and C.-Y. Wang, "Majority logic circuit minimization using node addition and removal," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 642–655, Mar. 2022.

[22] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *Proc. 24th International Conf. VLSI Design*, Jan. 2011, pp. 346–351.

[23] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," in *Proc. EDSSC*, Mar. 2010, pp. 1–4.

[24] Y.-A. Lai, C.-C. Lin, C.-C. Wu, Y.-C. Chen, and C.-Y. Wang, "Efficient synthesis of approximate threshold logic circuits with an error rate guarantee," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 773–778.

[25] C.-T. Lee, Y.-T. Li, Y.-C. Chen, and C.-Y. Wang, "Approximate logic synthesis by genetic algorithm with an error rate guarantee," in *Proc. 28th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2023, pp. 146–151.

[26] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1760–1771, Sep. 2013.

[27] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Proc. Design, Autom. Test Europe Conf. Exhib. (DATE)*, 2014, pp. 1–4.

[28] J. Ma and S. Hashemi, "Approximate logic synthesis using BLASYS," in *Proc. Workshop Open-Source EDA Technol.*, no. 5, 2019.

[29] D. S. Marakkalage, E. Testa, H. Riener, A. Mishchenko, M. Soeken, and G. De Micheli, "Three-input gates for logic synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 10, pp. 2184–2188, Oct. 2021.

[30] C. Meng, A. Mishchenko, W. Qian, and G. De Micheli, "Efficient resubstitution-based approximate logic synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 44, no. 6, pp. 2040–2053, Jun. 2025.

[31] C. Meng, W. Qian, and A. Mishchenko, "ALSRAC: Approximate logic synthesis by resubstitution with approximate care set," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[32] C. Meng et al., "SEALS: Sensitivity-driven efficient approximate logic synthesis," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 439–444.

[33] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2013, pp. 779–786.

[34] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 258–261.

[35] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of approximate sequential circuits," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2014, pp. 1–6.

[36] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, and L. Pozzi, "Partition and propagate: An error derivation algorithm for the design of approximate circuits," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2019, pp. 1–6.

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

[38] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 5, pp. 1694–1702, May 2017.

[39] L. Sekanina, "Evolutionary algorithms in approximate computing: A survey," *J. Integr. Circuits Syst.*, vol. 16, no. 2, pp. 1–12, Aug. 2021.

[40] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *Proc. Design, Autom. Test Eur.*, Mar. 2011, pp. 1–6.

[41] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 474–479.

[42] S. Su et al., "VECBEE: A versatile efficiency–accuracy configurable batch error estimation method for greedy approximate logic synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 5085–5099, Nov. 2022.

[43] S. Su, Y. Wu, and W. Qian, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[44] K. S. Tam, C.-C. Lin, Y.-C. Chen, and C.-Y. Wang, "An efficient approximate node merging with an error rate guarantee," in *Proc. 26th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2021, pp. 266–271.

[45] Z. Vasicek and L. Sekanina, "Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware," *Genetic Program. Evolvable Mach.*, vol. 12, no. 3, pp. 305–327, Sep. 2011.

[46] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, Jun. 2015.

[47] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Proc. DATE*, Mar. 2013, pp. 1367–1372.

[48] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Proc. DAC Design Autom. Conf.*, Jun. 2012, pp. 796–801.

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

[50] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Proc. Des. Autom. Conf. (DAC)*, 2016, pp. 1–6.

[51] Y. Xia, Z. Li, G. G. Yen, and H. Xia, "A knee-guided evolutionary algorithm based navigation approach for mobile robots in intelligent manufacturing scenarios," *IEEE Trans. Autom. Sci. Eng.*, vol. 22, pp. 582–593, 2024.

[52] K. Xu, D. Zhang, J. An, L. Liu, L. Liu, and D. Wang, "GenExp: Multi-objective pruning for deep neural network based on genetic algorithm," *Neurocomputing*, vol. 451, pp. 81–94, Sep. 2021.

[53] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," Microelectron. Center North Carolina (MCNC), Res. Triangle Park, NC, USA, 1991.

[54] Y. Zhou, G. G. Yen, and Z. Yi, "A knee-guided evolutionary algorithm for compressing deep neural networks," *IEEE Trans. Cybern.*, vol. 51, no. 3, pp. 1626–1638, Mar. 2021.

[55] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *Proc. ISIC*, Oct. 2009, pp. 69–72.

Yi-Ting Li received the B.S. degree from the Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. He is currently pursuing the Ph.D. degree with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

His research interests include logic synthesis, optimization, and verification for VLSI designs.

Ihao Chen received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, in 1981, and the M.S. and Ph.D. degrees from the Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, USA, in 1985 and 1988, respectively.

He founded Incentia Design Systems Inc., in 1998, where he has served as the President, CEO, and CTO since then.

Yung-Chih Chen (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2003, 2005, and 2011, respectively.

He is currently an Associate Professor with the Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. His current research interests include logic synthesis, design verification, and design automation for emerging technologies.

Chun-Yao Wang (Senior Member, IEEE) received the Ph.D. degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2002.

Since 2003, he has been an Assistant Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu, where he is currently a Distinguished Professor. His current research interests include logic synthesis, optimization, and verification for very large-scale integrated/system-on-chip designs and emerging technologies.