An Efficient Approximate Node Merging with an Error Rate Guarantee

Kit Seng Tam, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang

ABSTRACT

Approximate computing is an emerging design paradigm for error-tolerant applications. e.g., signal processing and machine learning. In approximate computing, the area, delay, or power consumption of an approximate circuit can be improved by trading off its accuracy. In this paper, we propose an approximate logic synthesis approach based on a node-merging technique with an error rate guarantee. The ideas of our approach are to replace internal nodes by constant values and to merge two similar nodes in the circuit in terms of functionality. We conduct experiments on a set of IWLS 2005 and MCNC benchmarks. The experimental results show that our approach can reduce area by up to 80%, and 31% on average. As compared with the state-of-the-art method, our approach has a speedup of 51 under the same 5% error rate constraint.

ACM Reference Format:
Kit Seng Tam, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang. 2021. An Efficient Approximate Node Merging with an Error Rate Guarantee. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431550

1 INTRODUCTION

As semiconductor technology advances, the number of transistors in VLSI designs grows exponentially. Thus, high power consumption has become a major challenge for designers. One possible solution to this challenge is to minimize the designs while preserving the functionality as much as possible. In the meantime, many error-tolerant applications, such as multimedia processing or machine learning, etc., are also emerging. Thus, approximate computing [4] was proposed as a new design paradigm recently. Approximate computing trades off circuits' accuracies for achieving smaller areas, delays, or power consumptions. Many previous works have demonstrated the effectiveness of this design paradigm shift in different design levels ranging from algorithm [5][20], architecture [8][9], logic [1][17], and transistor [6] levels.

This work is supported in part by the Ministry of Science and Technology of Taiwan under MOST 106-2221-E-007-111-MY3, MOST 108-2218-E-007-061, MOST 109-2221-E-007-082-MY2, MOST 109-2221-E-155-047-MY2, and MOST 109-2224-E-007-005.
K. S. Tam, C.-C. Lin, and C.-Y. Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 30013, R.O.C. Y.-C. Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan 32003, R.O.C.

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431550

On the logic level, approximate computing constructs a circuit that approximately implements its function. Many previous works focused on datapath designs, such as adders [10][26] or multipliers [12][14]. Recently, approximate logic synthesis, which synthesizes an approximate circuit from the original one under the given error constraint, was proposed [15][21][22][23][25]. In [15], Lai *et al.* proposed a threshold logic network optimization method with a hybrid cost function. In [22], Venkataramani *et al.* proposed to identify signal pairs in the circuit with similar functionality, which requires a significant amount of computation, and replace one with the other. Wu *et al.* proposed to shrink nodes in a Boolean network by approximating their factored-form expressions [23]. Yao *et al.* proposed to apply approximate disjoint bi-decomposition to the nodes to simplify the circuits [25]. Su *et al.* proposed a more accurate batch error estimation to improve the existing approximate logic synthesis flows [21]. However, these previous works conduct many approximate modifications for obtaining the best optimization result in an operation, which is time-consuming. Thus, in this work, we propose an efficient two-phase approximate logic synthesis based on node merging.

The node merging is a logic optimization technique without approximation. However, here we adopt it for approximate logic synthesis for minimizing errors. Our approach has two phases. In the first phase, we selectively replace a node in the circuit with a constant 0 or 1 based on the magnitude of 1's probability of the node. This magnitude of 1's probability for the replacement is a user-defined parameter. In the second phase, we further replace a given target node n_t with its substitute node n_s, where n_t and n_s are with similar functionalities. In summary, the proposed approach is to synthesize an approximate circuit by merging nodes such that the required error rate constraint is met. We demonstrate that our approach is much more efficient than the state-of-the-art [21]. The experimental results show that our approach achieved an average of 31% area reduction and had a speedup of 51 compared with [21] under the same 5% error rate constraint.

2 PRELIMINARIES

2.1 Error Metrics

To evaluate the error of an approximate circuit, several error metrics, such as bit-flip error, error magnitude, and error rate have been proposed [7]. Bit-flip error refers to the number of incorrect bits in the approximate circuit, which is relevant to memory address approximation. Error magnitude refers to the maximal numerical deviation in an approximate circuit's outputs. Error rate refers to the ratio of the number of input patterns that produces incorrect outputs in an approximate circuit. Many previous works [10][12][15][21][22][23][25] used the error rate as the error metric, while few works [17][19] used the error magnitude. Since the error rate is the most commonly used metric among these error metrics, in this work, we adopt the error rate as the error metric.

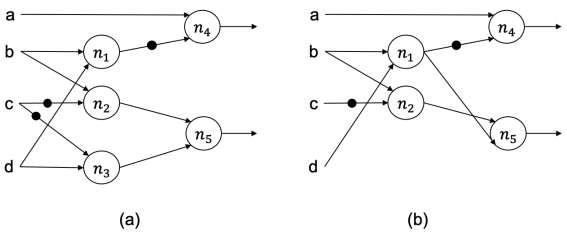

Figure 1: An example for presenting the node-merging approach. (a) The original circuit. (b) The resultant circuit after replacing n_3 by n_1.

2.2 Background

An *input-controlling value* of a gate g determines the output value of g no matter what the value of the other input is. An *input-noncontrolling value* of g is opposite to its input-controlling value. For an AND gate, its input-controlling value is 0 and its input-noncontrolling value is 1.

The *dominators* [11] of a gate g are a set of gates G that all the paths from g to any POs must pass through. The *side inputs* of G are the inputs of G that are not in the transitive fanout cone of g.

A *stuck-at fault* in VLSI testing is a fault model used to represent a manufacturing defect on wires or gates in digital circuits. A *stuck-at 0* (*sa*0) or *stuck-at 1* (*sa*1) fault on a faulty wire or faulty gate indicate that the signal on a faulty wire (gate) is stuck at a fixed logic value "0" or "1", respectively. A *stuck-at fault test* is a process to search for a test pattern that can generate the different values at any PO to distinguish a faulty circuit with a stuck-at fault from a fault-free one. A test pattern needs to *activate* the fault effect and *propagate* the fault effect to any PO. If there exists no test pattern that can both *activate* and *propagate* the fault effect to any PO, the fault is an *untestable* fault.

The *mandatory assignments* (MAs) are the unique value assignments to some nodes necessary to generate a test pattern for detecting a fault in the circuit. Consider a stuck-at fault on a gate g, the MAs are obtained by setting g to the fault-activation value and by setting the side inputs of the dominators of g to the input-noncontrolling values. By performing logic implications forward and backward from these MAs, more MAs can be inferred. If the MAs of a stuck-at v (*sa*v) fault on a wire are inconsistent, it means that no test pattern exists for this fault. Therefore, this fault is untestable and g can be replaced by the faulty value v.

2.3 Node-Merging Approach

The node-merging (NM) approach [2][3] is a logic optimization technique that identifies node mergers for obtaining a minimized circuit considering observability don't cares (ODCs). NM modeled the process of merging two nodes as a misplaced-wire error in the circuit and discussed the detection of this error. We use the example in Fig. 1 to demonstrate the NM [2][3]. The circuit in Fig. 1(a) is presented in And-Inverter-Graphs (AIGs) [18], where a, b, c, and d are primary inputs (PIs), nodes $n_1 \sim n_5$ are two-input AND gates, and a dot on an edge is an inverter. In this circuit, n_1 and n_3 are not functionally equivalent, merging them (i.e., creating a misplaced-wire error) may affect the overall functionality of the circuit. However, we observed that the values of n_1 and n_3 only differ when $b = c$ and $d = 1$. Since $b = c$ implies $n_2 = 0$, and $n_2 = 0$ is an input-controlling value to n_5, $n_2 = 0$ can block the error effect of merging n_3 with n_1. Thus, this misplaced wire error is undetectable, and merging nodes n_3 with n_1 will not change the overall functionality of the circuit.

To detect the error of merging two nodes, the input pattern has to cause different values on n_t and n_s for activating and propagating the error effect to any PO. If there is no input pattern that can detect the error, merging n_t with n_s is safe from the viewpoint of circuit's overall functionality. NM [2][3] proposed a sufficient condition about merging n_t and n_s as stated in Condition 1.

Condition 1 [2][3]: Let f denote an error of replacing n_t with n_s. If $n_s = 1$ and $n_s = 0$ are MAs for the stuck-at 0 and stuck-at 1 fault test on n_t, respectively, f is undetectable.

Let us briefly explain the effectiveness of Condition 1. The error f in Condition 1 represents the merging of n_t and n_s. Since generating different values for n_t and n_s is equivalent to activating the error effect, generating $n_s = 0$ is necessary for testing the *sa*0 fault on n_t (n_t has to be 1 for activating *sa*0 fault). However, if $n_s = 1$ is also an MA for the *sa*0 fault on n_t, the *sa*0 fault on n_t is untestable due to the contradiction on the value of n_s. In the same way, if $n_s = 0$ is also an MA for the *sa*1 fault on n_t, the *sa*1 fault on n_t is untestable. Since the error of replacing n_t with n_s, where we care about the different values of n_t and n_s only, cannot be detected by testing the *sa*0 and *sa*1 faults on n_t, f is undetectable.

Based on Condition 1, the process of identifying the node mergers is to compute the MAs of the *sa*0 and *sa*1 fault tests on n_t. We also use the circuits in Fig. 1 to demonstrate the NM algorithm. Suppose n_3 is a target node n_t, we would like to find its substitute nodes n_s. We compute the MAs of the *sa*0 and *sa*1 fault tests by setting n_3 to the fault-activating value and the side inputs of n_3's dominators to the fault-propagating values. The MAs of the *sa*0 fault test on n_3 are $\{n_3 = 1, n_2 = 1, \boldsymbol{d = 1}, c = 0, b = 1, \boldsymbol{n_1 = 1}, n_4 = 0, \boldsymbol{n_5 = 1}\}$ and that of the *sa*1 fault test on n_3 are $\{n_3 = 0, n_2 = 1, \boldsymbol{d = 0}, c = 0, b = 1, \boldsymbol{n_1 = 0}, \boldsymbol{n_5 = 0}\}$. As a result, d and n_1 satisfy the requirement of Condition 1, and can be chosen as the n_s. However, although n_5 also satisfies the requirement of Condition 1, n_5 will not be chosen as an n_s. This is because n_5 is in the transitive fanout cone of the target node n_3. If n_3 is replaced by n_5, a cyclic combinational circuit occurs, which is not allowed in that work.

3 PROPOSED APPROACH

In this section, we first present the proposed approximate node merging, which consists of two phases, *node to constant* and *node replacement*. Then, we present the flow of the proposed approach.

3.1 Node to Constant

By observing the circuit, we find that the functionalities of some nodes are very similar to constant 0 or 1. If we can easily find these nodes and replace them by constant 0 or 1, the circuit can be simplified effectively with fewer errors. Thus, the *node to constant* phase is proposed to focus on replacing nodes by constant 0 or 1. In other words, a node having a higher magnitude of 1's probability will be replaced by a constant 1. The magnitude of 1's probability p for the replacement can be determined by users.

To determine if a node is similar to a constant 0 or 1, we simulate a large number r of random patterns for estimating the 1's probability

Algorithm 1: Pseudo-code for Node to Constant Phase

```
Input: Original circuit C, error rate constraint ε1;
Output: An approximate circuit C_approx;
Initialization: error rate er = 0; C_approx = C; er_counter = 0;
  er_period = 5;
RandomSimulation(C, r);
for each node in C_approx in the DFS order from POs to PIs
    if (|1|/r ≥ p)
        Replaced by constant 1;
        NewC_approx = CleanFaninCone(C_approx);
        er_counter = er_counter + 1;
        if (er_counter == er_period)
            er = Compute_error(C, NewC_approx);
            er_counter = 0;
    else if (|1|/r ≤ 1 − p)
        Replaced by constant 0;
        NewC_approx = CleanFaninCone(C_approx);
        er_counter = er_counter + 1;
        if (er_counter == er_period)
            er = Compute_error(C, NewC_approx);
            er_counter = 0;
    if (er > 0.5ε1) er_period = 1;
    if (er < ε1) C_approx = NewC_approx;
    else break;
return C_approx;
```

of each node in the circuit. For each node in the circuit, we count the number of 1s at the node, denoted as $|1|$, after simulating r patterns. If one node is with the probability of $\frac{|1|}{r} \geq p$, the node will be replaced by a constant 1. If one node is with the probability of $\frac{|1|}{r} \leq 1-p$, the node will be replaced by a constant 0.

Algorithm 1 is the pseudo-code of *node to constant* phase. The inputs to the algorithm are the original circuit C and the error rate constraint ε_1 in this phase. The output of the algorithm is an approximate circuit C_{approx} with an error rate less than ε_1. First, we initialize the error rate of approximate circuit er to zero and the period of checking the error rate er_{period} to five. Then, we simulate r random patterns[1] to obtain the number of 1s at each node. We iteratively examine a node from the POs to the PIs in the depth-first search (DFS) order and replace it with a constant if applicable. The reason for adopting the DFS order is that we can also remove the single-fanout fanin (SFoFi) nodes in the fanin cone of the replaced node for effectively reducing the circuit. Then the error rate of the new approximate circuit is estimated by using $10r$ random patterns. Since this phase allows more errors and *node to constant* operation usually does not create many errors, estimating the error rate after every *node to constant* operation is not necessary. To elevate the efficiency of our approach, we compute the error rate every five times of *node to constant* operations if the error rate is smaller than $0.5\varepsilon_1$; otherwise, we compute the error rate after every *node to constant* operation. If the error rate of approximate circuit is less than ε_1, the algorithm proceeds to the next iteration; otherwise, the approximate circuit in the last iteration is returned as the output.

[1]The number of random patterns r for simulating the circuit to obtain the number of 1s at each node is a user-defined parameter. In our approach, we set r to 10,000.

3.2 Node Replacement

After running the *node to constant* phase, we conduct the *node replacement* phase, which aims to find the substitute node n_s to replace the target node n_t efficiently. The idea is that when n_t and n_s are with a high similarity, we replace n_t with n_s under accepting some errors. One possible naive method to find out the n_t-n_s pairs is to compare the signatures of the nodes after simulation. For example, assume that the signature of n_t = 110011 and n_s = 010011 after simulating six random patterns, we can consider they are a n_t-n_s pair. This idea is similar to that of the *node to constant* phase. However, this naive method cannot effectively find out the n_t-n_s pairs as further considering the error rate. That is, this naive method cannot observe the error effect caused by the replacement during the procedure of finding n_s, which is quite important for the last phase of this approach. Therefore, we propose the *node replacement*, which can effectively find out the n_t-n_s pairs based on NM [2][3].

As mentioned in Section 2, NM [2][3] proposed a sufficient condition for finding node mergers. If we arbitrarily choose an n_t-n_s pair, the replacement might cause two faults. The first one is denoted as f_{10}, which means $n_t = 1$ before the replacement and $n_t = 0$ after the replacement. The second one is denoted as f_{01}, which means $n_t = 0$ before the replacement and $n_t = 1$ after the replacement. Next, we explain the relationship between Condition 1 of NM [2][3] and these two faults caused by the replacement from the viewpoint of test pattern existence. In the rest of this paper, $MAs(n_t = sa0)$ denotes the MAs for the $sa0$ fault test on n_t, and $MAs(n_t = sa1)$ denotes the MAs for the $sa1$ fault test on n_t. With these notations, we can divide Condition 1 into two parts:

(1) $n_s = 1$ is an MA in $MAs(n_t = sa0)$.
(2) $n_s = 0$ is an MA in $MAs(n_t = sa1)$.

In Part (1), for detecting n_t $sa0$ fault, we can derive the $MAs(n_t = sa0)$, which includes $n_t = 1$. We collect all the patterns satisfying $MAs(n_t = sa0)$ in T_0. If we find an $n_s = 1$ that is in $MAs(n_t = sa0)$ and use it to replace n_t, all the input patterns in T_0 cause $n_t = n_s = 1$. Due to the same value of n_t and n_s in T_0, replacing n_t with n_s will not cause f_{10}. Similarly, in Part (2), for detecting n_t $sa1$ fault, we can derive the $MAs(n_t = sa1)$, which includes $n_t = 0$. We collect all the patterns satisfying $MAs(n_t = sa1)$ in T_1. If we find an $n_s = 0$ that is in $MAs(n_t = sa1)$ and use it to replace n_t, all the input patterns in T_1 cause $n_t = n_s = 0$. Due to the same value of n_t and n_s in T_1, replacing n_t with n_s will not cause f_{01}. In summary, a node with a value simultaneously satisfying Part (1) and Part (2) of Condition 1 will not cause both f_{10} and f_{01} faults. Thus, the node can be selected as an n_s for replacement without changing circuit's functionality.

However, if a node satisfies Part (1) but does not satisfy Part (2) of Condition 1, the input patterns in T_1 will detect f_{01}. Similarly, if a node with a value only satisfies Part (2) of Condition 1, the input patterns in T_0 will detect f_{10}. Therefore, in the *node replacement* phase, we consider to select a node satisfying either Part (1) or Part (2) as an n_s to replace the target node n_t. The next issue to be considered is judging which node is a better n_s as satisfying either Part (1) or Part (2).

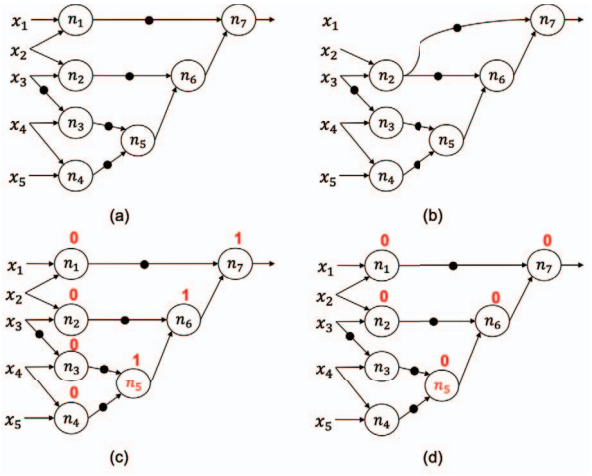

Figure 2: An example for demonstrating *node replacement*. (a) The original circuit. (b) The resultant circuit after replacing n_1 with n_2. (c) The value assignments of $MAs(n_5 = sa0)$. (d) The value assignments of $MAs(n_5 = sa1)$.

Algorithm 2: Pseudo-code for Node Replacement Phase

Input: Original circuit C, simplified circuit by *node to constant* phase C_{nc}, error rate constraint ε;
Output: An approximate circuit C_{approx};

```
Initialization: C_approx = C_nc; er_counter = 0; er_period = 3;
for each node n_t in C_approx in the DFS order from the POs to the PIs
    Compute MAs(n_t = sa0);
    Compute MAs(n_t = sa1);
    for each level l in C_approx in the ascending order from the PIs to the POs
        NA_0 = number of the l^th level's node assignments in MAs(n_t = sa0);
        NA_1 = number of the l^th level's node assignments in MAs(n_t = sa1);
        if (NA_0 = NA_1 = 0) continue;
        if (NA_0 < NA_1)
            n_s = node that has the lowest probability of 1 when n_t = 0 in MAs(n_t = sa0); break;
        else
            n_s = node that has the lowest probability of 0 when n_t = 1 in MAs(n_t = sa1); break;
    Replace n_t by n_s;
    NewC_approx = CleanFaninCone(C_approx);
    er_counter = er_counter + 1;
    if (er_counter == er_period)
        er = Compute_error(C, NewC_approx);
        er_counter = 0;
    if (er > 0.8ε) er_period = 1;
    if (er < ε) C_approx = NewC_approx;
    else break;
return C_approx;
```

We use an example in Fig. 2 to explain the satisfaction of either Part (1) or Part (2) in Condition 1. In the AIG of Fig. 2(a), suppose n_1 is the n_t. We first compute the $MAs(n_1 = sa0)$ and $MAs(n_1 = sa1)$, and they are $MAs(n_1 = sa0) = \{n_1 = 1, \boldsymbol{x_1 = 1}, \boldsymbol{x_2 = 1}, n_7 = 0, n_6 = 1, n_2 = 0, n_5 = 1, \boldsymbol{x_3 = 0}, n_3 = 0, n_4 = 0, \boldsymbol{x_4 = 0}\}$ and $MAs(n_1 = sa1) = \{n_1 = 0, n_7 = 1, n_6 = 1, n_2 = 0, n_5 = 1, n_3 = 0, n_4 = 0\}$, respectively. There are four MAs that are PIs in $MAs(n_1 = sa0)$, and $T_0 = \{x_1x_2x_3x_4x_5 = 11000, 11001\}$[2], which is represented as "1100-". Similarly, there is no PIs in $MAs(n_1 = sa1)$, then T_1 = "-----". The number of test patterns in T_0 is 2 and that in T_1 is $2^5 = 32$. Since in Part (1) and Part (2), the n_s for the replacement can make the corresponding fault untestable, we select the n_s that is with respect to a larger test pattern set. In this example, the size of T_1 is larger than T_0. Therefore, we choose an n_s satisfying Part (2) of Condition 1 to replace n_t.

On the other hand, if the size of T_0 and T_1 are the same, we choose an n_s from the part that has fewer assignments closest to the input side of circuit in the MA set. We use Figs. 2(c) and 2(d) to explain this situation. Suppose n_5 is the n_t, Fig. 2(c) shows the assignments in $MAs(n_5 = sa0)$. There are four assignments (n_1, n_2, n_3, and n_4) in $MAs(n_5 = sa0)$ that are closest to the input side of circuit. Similarly, Fig. 2(d) shows the assignments in $MAs(n_5 = sa1)$, but there are only two assignments (n_1 and n_2) that are closest to the input side of circuit. In this situation, we will choose the n_s from the set satisfying Part (2) of Condition 1.

After determining the source of n_s, either from Part (1) or Part (2), we next select an n_s to replace n_t. In the example of $n_t = n_1$, there are three n_s ($n_2 = 0$, $n_3 = 0$, and $n_4 = 0$) satisfying Part (2) of Condition 1. We will select the node that might cause fewer f_{10} faults after the replacement as the n_s. Therefore, we conduct another round of random simulation to obtain the probability of (n_t, n_s)=(1, 0). The probabilities of (n_t, n_s)=(1, 0) are $\frac{4}{32}$, $\frac{6}{32}$, or $\frac{6}{32}$ when $n_s = n_2$, n_3, or n_4, respectively. Since replacing n_1 with n_2 creates fewer f_{10} faults, n_2 is a better n_s to replace n_t. The resultant circuit of replacing n_1 with n_2 is shown in Fig. 2(b). If all the n_s are with the same probability, we randomly choose one n_s for the replacement.

Algorithm 2 is the pseudo-code of *node replacement* phase. The inputs to the algorithm are the original circuit C, the simplified circuit by the *node to constant* phase C_{nc}, and the error rate constraint ε. We initialize the period of checking the error rate er_{period} to three. Each node in C_{approx} is selected as n_t in the DFS order from the POs to the PIs. First, we compute the $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$. Then, we count the number of node assignments at the l^{th} level. The level of a node is the length of the shortest path from the PIs to the node. Note that when l is equal to 0, we use the number of PI assignments to represent the sizes of T_0 and T_1. If there does not exist the l^{th} level node assignment in $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$, we count the number of the next level's node assignment in $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$. Then, we select the MAs set with respect to n_s by choosing a smaller number between NA_0 and NA_1, where NA_0 and NA_1 are generally inversely proportional to the sizes of T_0 and T_1. Next, we choose the n_s that has the lowest probability of (n_t, n_s)=(v, v') when $n_s \in MAs(n_t = sav)$. Finally, we remove the SFoFi nodes in the fanin cone of n_t and compute the error rate er by conducting random simulation after

[2]The computation of all MAs for a stuck-at-fault test is an NP-complete problem [13]. Here we compute as many MAs as possible by a heuristic. Thus, T_0 and T_1 are not exact, and their sizes can be considered as the upper bound of test pattern number for testing the faults.

replacing n_t with n_s. Similar to the *node to constant* phase, the error rate is estimated by using $10r$ random patterns. If the error rate is smaller than 0.8ε, it is computed every three times of *node replacement* operation; otherwise, after every *node replacement* operation. If the error rate of approximate circuit has exceeded ε, the last legal approximate circuit is returned as the output.

3.3 Error Rate Estimation

As mentioned, we adopt the error rate as a metric to measure the quality of the approximate circuit. Since we do not simulate the input patterns exhaustively like most previous works, the error rate of an approximate circuit is estimated by EQ (1)

$$ErrorRate = \frac{|IncorrectPattern|}{|TotalSimulation|} \times 100\% \quad (1)$$

where $|TotalSimulation|$ represents the total number of random simulations, $|IncorrectPattern|$ is the number of simulations having incorrect outputs. At the end of the *node to constant* phase, we use $10r$ random patterns to determine if the error rate has exceeded ε_1. At the end of *node replacement* phase, we use another set of $10r$ random patterns to evaluate the final error rate ε. Note that in the later iterations of approximations, it is much easier for the error rate to violate the error rate constraint. Therefore, we heuristically set the different periods for checking the error rate at earlier and later iterations.

4 EXPERIMENTAL RESULTS

We implemented the proposed approach in C language within an ABC [27] environment. The experiments were conducted on an Intel Xeon E5-2650v2 2.60 GHz CentOS 6.7 platform with 64GBytes. The benchmarks are from IWLS 2005 [28] and MCNC [24]. Since approximately optimizing a design with a long critical path design causes an enormous error effect, the arithmetic multiplier was excluded from the benchmarks. The combinational portion of each benchmark was considered only and transformed into an AIG format. We used the error rate calculation tool [16] for computing the exact error rate of the approximated circuits.

We conducted three experiments. The first one is to present the circuit size reduction by setting different magnitude of 1's probability (p) in the *node to constant* phase. The second one is to compare the circuit size reduction and the CPU time among our approach, the state-of-the-art [21], and a naive method under a 5% error rate constraint. The last one is to demonstrate the effectiveness of our approach under different error rate constraints.

4.1 Probability p in Node to Constant Phase

Table 1 shows the result comparison in the *node to constant* phase when using different magnitudes of 1's probability p under 3% ($\varepsilon_1 = 0.6\varepsilon = 0.6 \times 5\%$) error rate constraint. Columns 1 ~ 3 list the information of benchmarks including names, the number of PIs and POs, and the number of nodes in each benchmark represented by AIG, respectively. Columns 4 ~ 7 list the percentages of the node count reduction (R.) of the four p values (99%, 98%, 97%, and 96%).

For example, the benchmark *alu4* has 14 PIs and 8 POs, and 1601 nodes. 23.24% of nodes in the circuit were removed by setting p to 98%. However, setting p to 99%, 97%, and 96% removed 21.24%, 16.61%, and 18.05% of nodes in the circuit, respectively. According to Table 1, setting p to 98% in our approach can remove 17.63% of nodes for all the benchmarks on average, which is the best among these four p values. When setting $p = 99\%$, since there are not many nodes satisfying this requirement, the amount of nodes that can be approximated to constant 0 or 1 is fewer. On the other hand, when we set $p = 96\%$, although many nodes can be approximated to constant 0 or 1, the caused larger error effect terminates this phase earlier. As a result, we heuristically set p to 98% in the *node to constant* phase to minimize the circuit size.

Table 1: The comparison of node count reduction among different magnitudes of 1's probability in the node to constant phase

Benchmark Information			Node Count Reduction (R.)(%)			
Name	|PI|/|PO|	|Node|	p = 99%	p = 98%	p = 97%	p = 96%
misex	25/18	91	29.67	29.67	29.67	0.00
c880	60/26	323	11.76	11.76	10.53	10.53
chkn	29/7	344	63.08	63.08	63.08	68.90
c1908	33/25	412	13.59	22.33	23.06	23.06
i9	88/63	541	0.00	0.00	0.92	0.92
c2670	233/140	694	17.58	18.16	18.88	0.00
simple_spi	148/144	815	9.82	18.90	18.90	9.45
c3540	50/22	941	4.25	4.25	3.08	0.00
dalu	75/16	1067	4.87	4.87	9.18	4.87
cps	24/109	1244	49.92	49.92	36.82	19.69
c5315	179/123	1415	1.91	1.91	0.00	0.00
c7552	207/108	1537	0.78	0.78	1.61	4.42
alu4	**14/8**	**1601**	**21.24**	**23.24**	**16.61**	**18.05**
s15850	611/684	2752	19.66	19.66	19.66	19.88
des_area	368/192	4391	1.57	0.98	0.43	0.00
s38417	1664/1742	8147	12.80	12.59	0.00	0.00
Average	—	1645	**16.41**	**17.63**	**15.78**	**11.24**

4.2 Circuit Size Reduction

In the second experiment, we compare the circuit size reduction using our approach against the state-of-the-art [21] and the naive method. In the naive method, the *node replacement* phase is changed to use the same idea of the *node to constant* phase when choosing the n_t-n_s pairs for the replacement.

In Table 2, Columns 4 ~ 6 list the percentages of the node count reduction (R.), error rate (E.), and the required CPU time measured in second. Columns 7 ~ 9 list the corresponding results of [21] and Column 10 lists the ratio of CPU time between [21] and our approach for each benchmark. Columns 11 ~ 14 list the corresponding results of the naive method. Since the benchmark s38417 is too large, the error rate calculation tool [16] cannot report the error rate of this benchmark. According to Table 2, the exact error rates for all the benchmarks in the three approaches are within 5%. The percentages of average node reduction for our approach and [21] are similar. However, our approach has a speedup of 51 under the same 5% error rate constraint considering all the benchmarks. The average speedup is 12 for all the benchmarks. It can be seen that the speedup of our approach is high for larger circuits, e.g., s38417, s15850. This indicates that our approach is more scalable than [21]. On the other hand, the naive method spent 208.72 seconds for removing 17.60% of nodes for all the benchmarks on average, while our approach spent 48.95 seconds for removing 31.06% of nodes on average.

In the last experiment, we demonstrate the effectiveness of our approach by setting 5% and 10% error rate constraints. Since the CPU time of the state-of-the-art [21] for 10% error rate constraint

Table 2: The comparison of experimental results among the state-of-the-art [21], the naive method, and our approach

Benchmark Information			Ours			Su's [21]				Naive			
Name	|PI|/|PO|	|Node|	R.(%)	E.(%)	Time (s)	R.(%)	E.(%)	Time (s)	Ratio	R.(%)	E.(%)	Time (s)	Ratio
misex	25/18	91	58.24	3.98	1.45	52.75	4.54	1.55	1.07	47.25	4.79	4.63	3.19
c880	60/26	323	17.34	4.29	3.71	14.86	4.09	2.59	0.70	7.12	5.00	31.58	8.51
chkn	29/7	344	80.81	4.83	5.69	80.52	4.91	9.68	1.70	72.38	3.48	23.41	4.11
c1908	33/25	412	60.92	4.05	3.41	61.65	3.66	8.43	2.47	39.32	3.71	17.64	5.17
i9	88/63	541	1.66	3.71	5.94	2.40	4.78	2.17	0.37	1.29	3.71	6.90	1.16
c2670	233/140	694	22.19	3.33	6.35	29.68	4.47	51.17	8.06	18.16	4.48	25.57	4.03
simple_spi	148/144	815	20.86	5.00	12.64	19.75	4.80	20.67	1.64	10.18	3.62	17.45	1.38
c3540	50/22	941	11.58	4.81	30.79	8.18	4.76	10.16	0.33	1.38	4.79	28.95	0.94
dalu	75/16	1067	33.83	5.00	38.56	34.68	4.90	47.95	1.24	0.84	3.56	13.36	0.35
cps	24/109	1244	68.65	4.71	25.41	70.50	4.90	110.49	4.35	20.02	4.71	10.87	0.43
c5315	178/123	1415	6.15	4.87	9.41	1.84	3.88	48.86	5.19	3.96	4.93	151.23	16.07
c7552	207/108	1537	7.29	4.11	29.52	9.04	1.86	18.25	0.62	1.43	4.64	36.71	1.24
alu4	14/8	1601	44.28	4.13	26.72	31.92	1.23	32.64	1.22	20.67	4.57	37.50	1.40
s15850	611/684	2752	32.34	4.89	136.92	35.17	4.98	2546.44	18.60	24.02	5.00	207.63	1.52
des_area	368/192	4391	5.26	3.01	185.74	7.79	4.98	775.06	4.17	0.14	4.77	83.26	0.45
s38417	1664/1742	8147	25.56	—	260.97	25.19	—	36819.18	141.09	13.42	—	2642.78	10.13
Average	—	1645	**31.06**	—	**48.95**	**30.37**	—	**2531.58**	**12.05**	**17.60**	—	**208.72**	**3.76**
Ratio	—	—	—	—	**1**	—	—	**51.72**	—	—	—	**4.26**	—

Table 3: The comparison of experimental results under different error rate constraints in our approach

Benchmark Information			$\varepsilon = 5\%$		$\varepsilon = 10\%$	
Name	|PI|/|PO|	|Node|	R.(%)	Time (s)	R.(%)	Time (s)
misex	25/18	91	58.24	1.45	71.43	1.63
c880	60/26	323	17.34	3.71	21.05	7.91
chkn	29/7	344	80.81	5.69	89.53	9.24
c1908	33/25	412	60.92	3.41	62.86	8.03
i9	88/63	541	1.66	5.94	6.28	6.56
c2670	233/140	694	22.19	6.35	35.45	7.98
simple_spi	148/144	815	20.86	12.64	25.15	27.55
c3540	50/22	941	11.58	30.79	17.22	35.3
dalu	75/16	1067	33.83	38.56	46.30	64.67
cps	24/109	1244	68.65	25.41	69.77	26.49
c5315	178/123	1415	6.15	9.41	8.48	11.54
c7552	207/108	1537	7.29	29.52	11.39	48.30
alu4	14/8	1601	44.28	26.72	52.34	87.59
s15850	611/684	2752	32.34	136.92	35.32	152.24
des_area	368/192	4391	5.26	185.74	8.75	146.93
s38417	1664/1742	8147	25.56	260.97	26.70	580.99
Average	—	1645	**31.06**	**48.95**	**36.75**	**76.43**

exceeded the time limit, 10 hours, we cannot list their results here. In Table 3, Columns 4 ~ 5 and Columns 6 ~ 7 list the results under 5% and 10% error rate constraints, respectively. According to Table 3, our approach for 10% error rate constraint achieved more circuit size reduction on average with some CPU time overhead.

5 CONCLUSION

In this paper, we propose an efficient node merging approach to synthesize the approximate circuits under the error rate constraint. The main ideas include changing nodes to constant nodes, and replacing the target nodes by the substitute nodes with a high similarity. The experimental results demonstrate that our approach has achieved the similar quality of approximate circuit as compared to the state-of-the-art, while having a significant speedup.

REFERENCES

[1] L. Chakrapani *et al.*, "A Probabilistic Boolean Logic for Energy Efficient Circuit and System Design," *Proc. ASP-DAC*, 2010, pp. 628-635.

[2] Y.-C. Chen *et al.*, "Fast Detection of Node Mergers Using Logic Implications," *Proc. ICCAD*, 2009, pp. 785-788.

[3] Y.-C. Chen *et al.*, "Fast Node Merging with Don't Cares Using Logic Implications," *IEEE TCAD*, 2010, pp. 1827-1832.

[4] V. Chippa *et al.*, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," *Proc. DAC*, 2013, pp. 1-9.

[5] V. Chippa *et al.*, "Dynamic Effort Scaling: Managing the Quality-Efficiency Trade-off," *Proc. DAC*, 2011, pp. 603-608.

[6] V. Gupta *et al.*, "IMPACT: IMPrecise adders for low-power Approximate CompuTing," *Proc. ISLPED*, 2011, pp. 409-414.

[7] J.Han *et al.*, "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design," *Proc. ETS*, 2013, pp. 1-6.

[8] K. He, *et al.*, "Controlled Timing-Error Acceptance for Low Energy IDCT Design," *Proc. DATE*, 2011, pp. 1-6.

[9] M. Imani *et al.*, "Resistive Configurable Associative Memory for Approximate Computing," *Proc. DATE*, 2016, pp. 1327-1332.

[10] Y. Kim *et al.*, "An Energy Efficient Approximate Adder with Carry Skip for Error Resilient Neuromorphic VLSI Systems," *Proc. ICCAD*, 2013, pp. 130-137.

[11] T. Kirkland *et al.*, "A Topological Search Algorithm for ATPG," *Proc. DAC*, 1987, pp. 502-508.

[12] P. Kulkarni *et al.*, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," *Proc. VLSID*, 2011, pp. 346-351.

[13] W. Kunz *et al.*, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation for Digital Circuits," *Proc. Int. Test Conf.*, 1992, pp. 816-825.

[14] K. Y. Kyaw *et al.*, "Low-Power High-Speed Multiplier for Error-Tolerant Application," *Proc. EDSSC*, 2010, pp. 1-4.

[15] Y.-A Lai *et al.*, "Efficient Synthesis of Approximate Threshold Logic Circuits with an Error Rate Guarantee," *Proc. DATE*, 2018, pp. 773-778.

[16] N. Lee *et al*, "Towards Formal Evaluation and Verification of Probabilistic Design," *IEEE Trans. on Computers*, 2018, pp. 1202-1216.

[17] J. Miao *et al.*, "Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints," *Proc. ICCAD*, 2013, pp. 779-786.

[18] A. Mishchenko, "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits," *IEEE Trans. Electronic Computers*, 1963, pp. 198-223.

[19] I. Scarabottolo, "Partition and Propagate: an Error Derivation Algorithm for the Design of Approximate Circuits," *Proc. DAC*, 2019, pp. 1-6.

[20] N. Shanbhag, "Reliable and Energy-Efficient Digital Signal Processing," *Proc. DAC*, 2002, pp. 830-835.

[21] S. Su *et al.*, "Efficient Batch Statistical Error Estimation for Iterative Multi-level Approximate Logic Synthesis," *Proc. DAC*, 2018, pp. 1-6.

[22] S. Venkataramani *et al.*, "Substitute-and-Simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits," *Proc. DATE*, 2013, pp. 1367-1372.

[23] Y. Wu *et al.*, "An Efficient Method for Multi-level Approximate Logic Synthesis under Error Rate Constraint," *Proc. DAC*, 2016, pp. 1-6.

[24] S. Yang, "Logic Synthesis and Optimization Benchmarks," Microelectronics Center of North Carolia, Tech. Rep., 1991.

[25] Y. Yao *et al.*, "Approximate Disjoint Bi-decomposition and Its Application to Approximate Logic Synthesis," *Proc. ICCD*, 2017, pp. 517-524.

[26] N. Zhu *et al.*, "An Enhanced Low-Power High-Speed Adder for Error-Tolerant Application," *Proc. ISIC*, 2009, pp. 69-72.

[27] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification* [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc

[28] http://iwls.org/iwls2005/benchmarks.html