# An Efficient Method for Multi-level Approximate Logic Synthesis under Error Rate Constraint

Yi Wu and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {eejessie, qianwk}@sjtu.edu.cn

## ABSTRACT

Approximate computing is an emerging design paradigm targeting at error-tolerant applications. It trades off accuracy for improvement in hardware cost and energy efficiency. In this paper, we propose a novel approach for multi-level approximate logic synthesis under error rate constraint. The basic idea of our approach is to pick nodes in a Boolean network and shrink them by approximating their factored-form expressions. We propose two different algorithms to implement the basic idea. The first algorithm iteratively picks the most effective node at present to shrink. Its drawback lies in that it may need a large number of iterations. To overcome this drawback, the second algorithm formulates a knapsack problem to pick multiple nodes for shrinking simultaneously. It is still iterative, but the number of iterations is greatly reduced. We apply the two algorithms to MCNC benchmarks and arithmetic circuits including adders and multipliers. The experimental results demonstrated that our algorithms perform better in area saving and are 1.7 and 5.9 times faster, respectively, compared with the state-of-the-art approach.

## CCS Concepts

•**Hardware → Combinational synthesis;**

## Keywords

Logic synthesis, Approximate computing, Inexact circuit

## 1. INTRODUCTION

As the size of CMOS transistors reaches nano-scale, reducing the energy consumption has become a key issue in VLSI design. One effective solution to this problem is to simplify the circuit as much as possible so that it has less area, delay, and power consumption. Meanwhile, many applications that exhibit an inherent tolerance to errors in computation, such as multimedia, signal processing, and data mining, become more widely used with the prevalence of mobile computing and embedded systems. Given this context, a novel computing paradigm called *approximate computing* was proposed, which deliberately trades off accuracy for energy efficiency. By allowing a small amount of error, the design space is significantly enlarged, which may lead to much more energy-efficient designs. Many studies have demonstrated the effectiveness of this paradigm at different design levels ranging from algorithms, architectures, to logic and transistors [1].



*DAC '16, June 05-09, 2016, Austin, TX, USA*

DOI: http://dx.doi.org/10.1145/2897937.2897982

At logic level, approximate computing seeks to derive a circuit that not exactly but *approximately* implements the target function. Many pioneering works in this area focused on the manual design of approximate versions of arithmetic circuits, such as adders [2, 3] and multipliers [4, 5]. In recent years, approximate logic synthesis (ALS), which automatically synthesizes an approximate circuit for a specified Boolean function under the given error constraints, was proposed as a more promising approach since it allows larger design space exploration. Several efforts have been proposed to handle the ALS problem [1, 6–10].

To evaluate the quality of an approximate design, several error metrics have been adopted, among which error rate and error magnitude are the most commonly used ones. Error rate refers to the ratio of the number of input vectors that produce incorrect outputs to the total number of input vectors. Error magnitude refers to the maximal numerical deviation of an incorrect output from a correct one.

In this work, we propose a novel method for approximate logic synthesis of multi-level combinational circuits under the error rate constraint. Traditional multi-level logic synthesis usually includes two phases: a technology-independent synthesis phase and a technology mapping phase [11]. Our method focuses on the first phase. In this phase, a combinational circuit is represented as a *Boolean network*, in which each node represents a Boolean function [11]. Simplifying the nodes in a Boolean network helps reduce the total area of the circuit. The traditional logic synthesis methods optimize circuits while always keep their functions unchanged.

However, in the proposed approach, when simplifying a node, we allow the functional change of the node. Specifically, we simplify a node by removing literals from its factored-form expression. The obtained expression will become different from the original function. Among all literals in a factored-form expression, we can flexibly choose how many of them as well as which of them to be removed. Therefore, a node usually has multiple approximate expressions.

To get a minimal-area approximate circuit satisfying the error rate constraint, we propose two different algorithms: single-selection algorithm and multi-selection algorithm. The single-selection algorithm picks a node and its associated approximate expression which provides the best simplification opportunity at each iteration. Although the optimization effect is guaranteed, it suffers from long runtime since it usually takes many iterations to stop. Therefore, we further propose the multi-selection algorithm. In each iteration of this algorithm, we formulate a variation of the basic 0/1 knapsack problem, which we call *multi-state knapsack problem*. Each node is treated as an candidate item to be selected. We apply dynamic programming technique to decide which nodes and their associated approximate expressions should be selected. This multi-selection technique reduces the iteration number greatly while only has a tiny quality loss compared to the single-selection algorithm.

In summary, our main contributions are as follows:

- We propose a novel method to approximate a circuit by simplifying the factored-form expressions of local nodes in a Boolean network. Since a node usually

has multiple approximate expressions, this provides us with more freedom in exploring a good solution.

- We propose to choose multiple nodes for simplification simultaneously, which significantly reduces the runtime. We formulate a multi-state knapsack problem to determine the optimal choice.
- We propose a dynamic-programming-based approach to solve the multi-state knapsack problem.

The rest of the paper is organized as follows. Section 2 reviews the prior related work. Section 3 describes the basic idea of our approach and the key operations that are used. Section 4 and 5 present the single-selection algorithm and the multi-selection algorithm, respectively. The experimental results are presented in Section 6. Finally, Section 7 concludes the paper.

## 2. RELATED WORK

Recently, different two-level and multi-level ALS approaches have been proposed. The approach in [7] targets at two-level circuits and considers only the error rate constraint. The work [1] further considers synthesizing two-level approximate circuits under both error rate and error magnitude constraints. In contrast, our work focuses on ALS for multi-level circuits.

Prior works on multi-level ALS include [6, 8–10]. The work [8] proposed to deliberately inject stuck-at-faults into the circuit to simplify it under a composite constraint on error rate and error magnitude. Our approach is more general than this method, since injecting stuck-at-faults can be viewed as a special case of our approach. The work [9] proposed to encode the error constraint as a function and exploit *external don't cares* (EXDC) to simplify the circuit. However, it is only applicable to constraint on error magnitude, which is different from the constraint we are considering. The method in [10] has the capability of synthesizing approximate multi-level logic circuits under both error rate and error magnitude constraints. Its solution to satisfy both constraints is to solve the error-rate-unconstrained ALS first and then iteratively refine the resultant circuit until the error rate is within the threshold. Although the method can be applied to handle the error rate constraint alone, the quality is not so good since it prioritizes first on meeting the error magnitude constraint.

To the best of our knowledge, the state-of-the-art approach for multi-level ALS under error rate constraint is the SASIMI approach proposed in [6]. Its basic idea is to identify signal pairs that take the same value with high probability and substitute one with the other. In comparison to SASIMI [6], our method is based on simplifying each local node in a Boolean network, which was demonstrated by experimental results to be as powerful as their substitution-based method. Furthermore, the locality enables our algorithm to make multiple changes simultaneously, which significantly improves the runtime. The capability of choosing multiple changes simultaneously is also a major difference than all of the previous works.

## 3. BASIC IDEA AND KEY OPERATIONS

In this section, we introduce the basic idea of our approach and the key operations that are used.

Traditional logic synthesis tools, such as MIS [12] and SIS [13], optimize a combinational Boolean network using two basic techniques: global optimization and local transformation. Among many local transformation operations, *simplification* [12] plays an important role because it takes advantage of the immediate environment of a node to remove redundancy in the network. Each node in a Boolean network is represented both in a sum-of-product (SOP) form and a factored form. In the simplification operation, the two-level logic minimizer ESPRESSO [14] is called for each node to derive a minimal SOP form. However, the obtained SOP

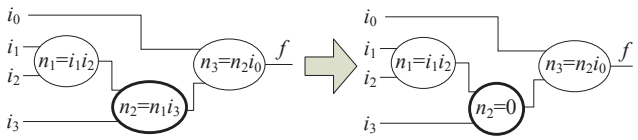


**Figure 1: A Boolean network with the original expression of $n_2$ replaced by the ASE $n_2 = 0$.**

form is accepted only if the number of literals in the corresponding factored form is reduced compared to the original factored form. This is because for multi-level Boolean networks, the factored form provides a better area estimate for the node [12].

Considering this fact, in our approach, we do not work on the SOP form for each node. Rather, we shrink nodes by directly manipulating their factored-form expressions.

**Definition 1** *By removing some literals from the original factored-form expression of a node, we obtain a simplified factored-form expression which is an approximation to the original expression. We call such an expression* **Approximate Simplified Expression (ASE)**.

If we use an ASE to replace the original factored-form expression of a node $n$ in a network, the Boolean function of $n$ will be modified. However, for the set of primary input (PI) patterns which cause an error at node $n$, only a subset of it can actually cause an error at the primary outputs (POs). The rest of them cannot propagate the error at the node $n$ to any PO. For example, consider a Boolean network with four PIs $i_0, i_1, i_2, i_3$ and one PO $f$ as shown in Fig. 1. The original factor-form expression of node $n_2$ is $n_2 = n_1 i_3$. If we use a constant 0 to replace its original expression, then two PI patterns 0111 and 1111 will cause an error at node $n_2$. However, the error at $n_2$ could be propagated to the PO $f$ only if $i_0$ takes the value 1. Thus, only the PI pattern 1111 will cause an error at PO. The other PI pattern 0111 will block the propagation of the error at $n_2$. To distinguish different PI patterns, we make the following definition.

**Definition 2** *If we use an ASE to replace the original expression of a node $n$, errors will occur at the output of $n$. We call a PI pattern that causes an error at node $n$ an* **Apparent Erroneous PI Pattern (AEPIP)** *of the ASE and a PI pattern that causes an error at the POs a* **Real Erroneous PI Pattern (REPIP)** *of the ASE. We call the error rate at node $n$ the* **Apparent Error Rate** *of the ASE and the error rate of the entire network the* **Real Error Rate** *of the ASE.*

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

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

On the one hand, the real error rate of an ASE accurately measures the contribution of an ASE to the error rate increase of the entire network. It will be used in the single-selection algorithm. On the other hand, the apparent error rate will play an important role in the multi-selection algorithm. Next, we will explain how we generate ASEs for a node in Section 3.1, how we compute apparent error rate of an ASE in Section 3.2, and how we estimate the real error rate of an ASE in Section 3.3.

### 3.1 Generating ASEs for a Node

An ASE for a node is obtained by deleting one or more literals from the original factored-form expression of the node. Assume a factored form has $N$ literals. Then there are in total $\binom{N}{M}$ different ways to remove $M$ literals from it. One

special case that needs attention is when $N = M$. There are two different simplifications: one is to make the node a constant 0 and the other is to make it a constant 1.

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

The two simplifications $n = 0$ and $n = 1$ are special because they completely delete the node $n$ from the network. Furthermore, after logic implication, some nodes in the transitive fan-in or fan-out cone of $n$ may also be simplified or deleted. Actually, this is just the way that [8] applies to do simplification. Their method injects a single stuck-at fault to the circuit iteratively, i.e., it chooses a node each time and makes it a constant 0 or 1. Therefore, the solution space of our method is larger than that of [8].

## 3.2 Computing Apparent Error Rate of an ASE

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

**Definition 3** *If we use an ASE to replace the original expression of a node $n$, errors will occur at the output of $n$. We call a local input pattern of $n$ that produces a wrong value at $n$ an* **Erroneous Local Input Pattern (ELIP)** *of the ASE.*

The apparent error rate of an ASE can be obtained by adding up the probabilities of all ELIPs of the ASE. For example, suppose the ELIPs of an ASE are 1001, 1010, and 1011. Assume their probabilities are 0.03, 0.01, and 0.02, respectively. Then, the apparent error rate of the ASE is 0.06. The probability of a given local input pattern can be obtained by running logic simulation on the entire circuit and counting the frequency that the specific pattern occurs. Note that to obtain the probabilities of all the local input patterns of interest for all the nodes, only one run of logic simulation is required.

## 3.3 Estimating Real Error Rate of an ASE

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

Our estimation method is based on the ELIPs of an ASE. For an internal node $n$ in a Boolean network, the ELIPs of an ASE account for the errors at the output of $n$. However, even in a well-optimized network, $n$ may have satisfiability don't cares (SDCs) and observability don't cares (ODCs) due to its surrounding nodes [15]. On the one hand, if an ELIP is an SDC of $n$, the ELIP cannot occur at all. On the other hand, if an ELIP is an ODC of $n$, the wrong value at $n$ caused by the ELIP cannot be propagated to the POs.

Therefore, to compute the real error rate of an ASE for a node $n$, we should ignore those ELIPs of the ASE that are either SDCs or ODCs of the node $n$. The remaining ELIPs will really introduce errors to the POs when they occur. We call them *non-don't-care ELIPs* of the ASE. Our method to estimate the real error rate of an ASE is to add up the probabilities of all the non-don't-care ELIPs. To obtain SDCs and ODCs of a node, we use the command "mfs" in MVSIS [16]. To speed up this process, we set the window size to $2 \times 2$ and choose a SAT-based computation method.

The obtained estimate indeed gives an upper bound on the real error rate of an ASE. There are two reasons for this:

1. Obtaining all the SDCs and ODCs of an internal node is computationally expensive. Instead, we just get a subset of the SDCs and ODCs.

2. For any local input pattern $V$ which is a non-don't-care ELIP of an ASE for a node $n$, not all the PI patterns that generate $V$ at the immediate inputs of $n$ can finally propagate the error at $n$ to the POs. For example, consider the circuit shown in Fig. 1. The local input pattern $(n_1, i_3) = (1, 1)$ is a non-don't-care ELIP of the ASE $n_2 = 0$ for node $n_2$. Although two PI patterns 0111 and 1111 could generate this local input pattern, only the pattern 1111 could propagate the error at $n_2$ to the PO. Therefore, the sum of the probabilities of all the non-don't-care ELIPs is always larger than or equal to the real error rate of the ASE.

Although our method just gives an upper bound on the real error rate of an ASE, it is much faster than getting the exact value. Furthermore, since it is an upper bound, once a change is selected, it is guaranteed to be accepted. Thus, we use this method to estimate the real error rate.

# 4. SINGLE-SELECTION ALGORITHM

In this section, we describe the single-selection algorithm. It iteratively picks one most effective node to shrink until the error rate exceeds the given threshold.

**Algorithm 1** Single-selection algorithm.

```
1: Inputs: The input network C and the threshold for error rate T.
2: Outputs: A min-area approximate network C' with error rate ≤ T.
3: Initialize: C' ⇐ C; current error rate ER ⇐ 0; error rate margin t ⇐ T;
4: while ER ≤ T do
5:    maxScore ⇐ −1;
6:    for each node n in the network C' do
7:       Find SDCs and ODCs of n and set their probabilities to 0;
8:       Estimate the real error rates of all ASEs for the node n;
9:       bestASE ⇐ max-score ASE of n with error rate ≤ t;
10:      if bestASE.score > maxScore then
11:         maxScoreNode ⇐ n; maxScore ⇐ bestASE.score;
12:   For the maxScoreNode in C', replace its original factored-form expression by its bestASE;
13:   Update ER; t ⇐ T − ER;
14: return C' in the last iteration;
```

The pseudo-code for the single-selection algorithm is shown in Algorithm 1. It takes a network $C$ and a given error rate threshold $T$ as inputs. We denote the approximate network as $C'$, which is initially set as the input network $C$. The variable $ER$ records the error rate of the network $C'$ at the start of each iteration. Another variable $t$ records the margin of the error rate threshold, which equals $T - ER$.

For each node $n$ in the network $C'$, Lines 7–9 find the best feasible ASE for $n$. An ASE is *feasible* if its real error rate is no more than the error rate margin $t$. In Line 7, we find SDCs and ODCs at the local inputs of the node $n$ and ignore them in the calculation of the real error rate of an ASE by setting their probabilities to 0. Then Line 8 estimates the real error rates of all the ASEs for the node $n$ by the method discussed in Section 3.3. For a node $n$ whose factored-form expression has $N$ literals, there are $\binom{N}{1} + \binom{N}{2} + \cdots + \binom{N}{N-1} + 2\binom{N}{N} = 2^N$ different ASEs in total. We estimate the real error rates for all these ASEs when $N < 5$. Note that in a well-optimized Boolean network, $N$ is usually less than 5. In

the case where $N \geq 5$, we only consider the ASEs obtained by removing less than 5 literals from the original expression as well as the constant 0 and constant 1 ASEs. Finally, among all the feasible ASEs of the node $n$, we pick the one with the maximum score as its best ASE, as shown in Line 9. The score of an ASE is defined as follows. Suppose an ASE $g$ is obtained by removing $l$ literals from the original factored-form expression of a node $n$ and the real error rate estimate of $g$ is $e$. Then, the score of $g$ is $l/e$.

After iterating over all the nodes, we pick out the node whose best ASE has the highest score among all the nodes and replace its original expression by its best ASE. In Line 13, we calculate the actual error rate $ER$ of the current network $C'$ and update the error rate margin $t$. The actual error rate is obtained by running logic simulation and counting the frequency that the POs are incorrect. When $ER$ exceeds the given threshold $T$, the loop terminates and the network $C'$ in the last iteration is returned as the final approximate network.

# 5. MULTI-SELECTION ALGORITHM

The single-selection algorithm may not be time-efficient when the iteration number is large. To speed up the whole process, we propose the multi-selection algorithm where multiple nodes are picked and shrunk in each iteration. There are two key questions related to this procedure: 1) How should we estimate the real error rate caused by the change of multiple nodes? 2) Which set of nodes and their associated ASEs should we pick to maximize the literal saving? We first answer these two questions in Section 5.1 and Section 5.2, respectively. Then, we show the overall flow of the multi-selection algorithm in Section 5.3.

## 5.1 Estimating Real Error Rate Caused by Multiple Changes

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

**Theorem 1** *Suppose $q$ nodes $n_1, \ldots, n_q$ are picked to make simultaneous change. Assume the original expression of node $n_i$ is replaced by an ASE $g_i$ and the apparent error rate of $g_i$ is $r_i$. Then the error rate of the new circuit is bounded by $r_1 + r_2 + \cdots + r_q$.*

We omit the proof due to the space limit. Based on the above theorem, in the multi-selection algorithm, we use the sum of the apparent error rates of the ASEs for all the nodes involved in the change to estimate the error rate increase of the new circuit. The apparent error rate of an ASE can be calculated by the method discussed in Section 3.2.

## 5.2 Selecting an Optimal Set of Nodes and Their ASEs

In the multi-selection algorithm, we need to decide which set of nodes we should pick and among all the feasible ASEs of each selected node, which one we should choose to maximize the literal saving.

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

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

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

The solution begins by filtering the states and items. We delete all the states with weight larger than the capacity $W$. If all the states of a candidate item are removed, then the item is also removed. For each of the remaining items, we will scan all of its remaining states and remove the states that are dominated by another state. For two states $s_1 = (w_{i1}, v_{i1})$ and $s_2 = (w_{i2}, v_{i2})$ of an item $c_i$, we say the state $s_1$ *dominates* the state $s_2$ if and only if $w_{i1} \leq w_{i2}$ and $v_{i1} \geq v_{i2}$. In this case, if the item $c_i$ is finally picked in the optimal solution, its state $s_2$ cannot be chosen as the associated state, because it is inferior to the state $s_1$. Thus, we remove the dominated state $s_2$.

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

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

In calculating $m[i, j]$ for $i \geq 1$, we need to consider all the states of the item $c_i$ with weight less than or equal to $j$. For example, in calculating $m[2, 8]$, we need to consider both the states $s_{21}$ and $s_{22}$ of the item $c_2$. If we select $c_2$ and pick its state $s_{21}$, then the maximum value is $m[1, 8 - 4] + 2 = 2 + 2 = 4$. If we select $c_2$ and pick its state $s_{22}$, then the maximum value is $m[1, 8 - 6] + 4 = 1 + 4 = 5$. We should also consider the case where the item $c_2$ is not selected. In this case, the maximum value is $m[1, 8] = 2$. Comparing the above three choices, we can see that selecting $c_2$ and its state $s_{22}$ gives the maximum possible value with weight less than or equal to 8 using up to the first 2 items. The maximum value is $m[2, 8] = 5$.

Once we have constructed the entire table, the lower right corner of the table gives the optimal value of the problem. The final selected items and their states can be back tracked from the optimal value. For our example, the optimal value is 6 and the optimal choice consists of item $c_1$ with its state as $s_{12}$ and item $c_2$ with its state as $s_{22}$.

**Table 1: Candidate items and their states.**

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

**Table 2: Solution of the multi-state knapsack problem shown in Table 1.**

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

## 5.3 Flow of the Multi-selection Algorithm

Finally, we show the flow of the multi-selection algorithm in Algorithm 2. The definitions of the variables are the same as those in Algorithm 1. In Lines 6–8, we iterate over all the nodes in network $C'$ and put all ASEs of each node in the set *ASE_record*. In Line 9, we apply *Knapsack_Solver*, which implements the solution to the knapsack problem discussed above, on *ASE_record* to choose the optimal set of nodes and their associated ASEs. The selected nodes and their ASEs are stored in *select_record*. In Lines 10–11, we simplify the network $C'$ based on the items in *select_record*.

Note that at the end of each iteration, we compute the real error rate of the network by simulation to guarantee that the error rate due to the introduced errors is still below the given threshold. Therefore, although the error rates of the ASEs are not obtained exactly, it will not cause violation to the specified error rate constraint.

**Algorithm 2** Multi-selection algorithm.

1: **Inputs:** The input network $C$ and the threshold for error rate $T$.
2: **Outputs:** A min-area approximate network $C'$ with error rate $\leq T$.
3: Initialize: $C' \Leftarrow C$; current error rate $ER \Leftarrow 0$; error rate margin $t \Leftarrow T$;
4: **while** $ER \leq T$ **do**
5: $ASE\_record \Leftarrow \varnothing$;
6: **for** each node $n$ in the network $C'$ **do**
7: Calculate the apparent error rates of all ASEs of $n$;
8: Find all ASEs of $n$ and store them in $ASE\_record$;
9: $select\_record \Leftarrow Knapsack\_Solver(ASE\_record)$;
10: **for** each node $n$ in *select_record* **do**
11: Use the selected $ASE$ to replace the original expression of $n$;
12: Update $ER$; $t \Leftarrow T - ER$;
13: **return** $C'$ in the last iteration;

# 6. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of our approach, we implemented the two algorithms in C++ and applied them to a set of benchmarks, including the circuits in MCNC benchmarks [17] and some arithmetic circuits, which were synthesized by Synopsys Design Compiler [18]. Table 3 lists the information of the benchmark circuits used in the experiments. The area and delay values were reported by SIS [13].

In our implementation, logic simulation was performed by the Verilog logic simulation tool Synopsys VCS [18]. We assumed all PI patterns were of equal probability. Each run of logic simulation applied 10000 randomly generated PI vectors, which was able to provide accurate enough error rates. After generating the approximate Boolean network, we further performed technology mapping, which was done by the logic synthesis tool SIS [13] using the MCNC generic standard cell library. All the experiments were conducted on a virtual machine running Linux operating system with 1GB memory. The host machine is a 3.1 GHz desktop.

For comparison, we also implemented the state-of-the-art approach SASIMI proposed in [6]. However, since our method does not consider the timing constraints of the circuits, for fair comparison, we also excluded the operations that handle timing constraints in the implementation of SASIMI so that it can optimize the area as much as possible. In SASIMI, gate downsizing is applied to further reduce the area. For our approach, since it works at the technology-independent synthesis phase, gate downsizing is not applicable yet. For fair comparison, we also excluded gate downsizing operation in the implementation of SASIMI.

**Table 3: Benchmark information.**

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

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

Fig. 2 shows the area saving for all the benchmarks by applying the single-selection algorithm. We can see that for each benchmark, as the error rate increases, the area saving increases as well. When the error rate threshold is 5%, for most benchmarks, their approximate circuits have $15\% \sim 35\%$ area saving. For c1908, the percentage of area saving even reaches 72%. Note that for benchmarks c2670, CLA32, and MUL8, when the error rate is 0, the area saving is larger than 0. The reason is because the single-selection algorithm is able to identify some redundancy which cannot be captured by the traditional logic synthesis tool. The plot of the area saving of the multi-selection algorithm is close to that of the single-selection algorithm. Due to the page limit, we do not show it here.

Comparison in area saving and runtime among different algorithms is presented in Table 4. The "SASIMI" column shows the results produced by our implementation of SASIMI [6]. The "single-selection" column and the "multi-selection" column list the results of the two algorithms proposed in this work, respectively. For each benchmark, we ran the experiments for seven different error rate thresholds:

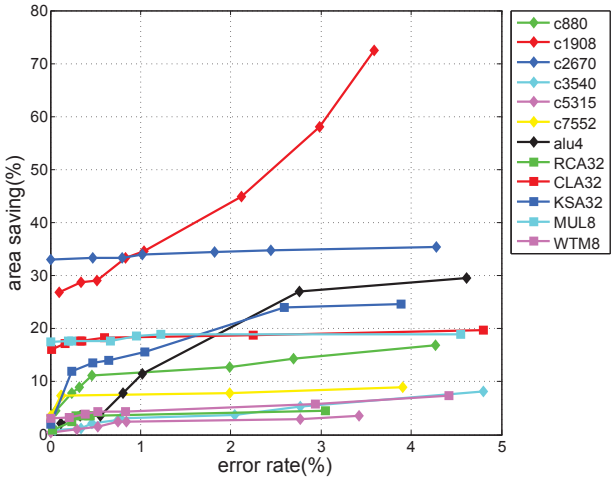


**Figure 2: Area saving of the single-selection algorithm.**

0.1%, 0.3%, 0.5%, 0.8%, 1%, 3%, and 5%. For each error rate threshold, the ratio of the area of the obtained approximate circuit to that of the original circuit was calculated. Then the average value of the seven ratios was computed and listed in the "area ratio" columns in Table 4. To compare the runtime, we also computed the average runtime in seconds over the seven experiments on different error rate thresholds and listed it in the "time/s" column.

**Table 4: Area saving and runtime comparisons between our approaches and the approach SASIMI in [6].**

| | SASIMI | | single-selection | | multi-selection | |
|---|---|---|---|---|---|---|
| circuit | area ratio | time/s | area ratio | time/s | area ratio | time/s |
| c880 | 0.896 | 154 | 0.893 | 93 | 0.893 | 48 |
| c1908 | 0.610 | 1090 | 0.595 | 394 | 0.598 | 181 |
| c2670 | 0.724 | 664 | 0.662 | 702 | 0.673 | 90 |
| c3540 | 0.975 | 393 | 0.966 | 172 | 0.965 | 77 |
| c5315 | 0.981 | 996 | 0.978 | 263 | 0.981 | 85 |
| c7552 | 0.948 | 2665 | 0.940 | 533 | 0.941 | 173 |
| alu4 | 0.892 | 645 | 0.878 | 1000 | 0.869 | 186 |
| RCA32 | 0.972 | 33 | 0.970 | 40 | 0.969 | 15 |
| CLA32 | 0.829 | 196 | 0.822 | 213 | 0.822 | 57 |
| KSA32 | 0.830 | 553 | 0.827 | 495 | 0.831 | 39 |
| MUL8 | 0.829 | 1095 | 0.819 | 223 | 0.826 | 151 |
| WTM8 | 0.959 | 249 | 0.953 | 168 | 0.956 | 57 |
| Geomean | 0.863 | 452 | 0.841 | 260 | 0.852 | 77 |

Comparing the three "area ratio" columns in Table 4, we find that the single-selection algorithm performs better in area saving than SASIMI for all the benchmarks and the multi-selection algorithm performs better than SASIMI for all the benchmarks except KSA32. The multi-selection algorithm performs slightly worse than the single-selection algorithm in terms of area saving. We believe the reason is that the error rate estimate in the single-selection algorithm is more accurate than that in the multi-selection algorithm and hence, more opportunities for area saving are explored.

As shown in Table 4, the major advantage of our approaches lies in runtime. On average, the speedups of the single-selection algorithm and the multi-selection algorithm over SASIMI are 1.7× and 5.9×, respectively. SASIMI is slower than ours because it performs pairwise comparisons of all the signals in the network. Thus, its runtime complexity is quadratic to the number of nodes in the network. In contrast, both of our approaches have runtime complexity linear to the number of nodes in the network.

Although the proposed approaches do not set timing constraint for the circuit, we observed that for all the benchmarks except CLA32 and for all the error rate thresholds, the obtained approximate circuits have smaller delays than the original circuits. The reason for this is because our approaches always shrink nodes to simpler forms and we do not apply global transformations which may reconstruct the circuit and increase the delay of the circuit. We also observed that the delays produced by our approaches and those produced by SASIMI are very close.

# 7. CONCLUSION

In this work, we presented a novel method to synthesize approximate circuits for error tolerant applications under the error rate constraint. The key idea of our approach is to shrink nodes in a network by approximating their factored-form expressions. We developed two different algorithms to obtain a min-area approximate version of the original circuit which satisfies the error rate bound. The first algorithm picks one most effective node and uses its best ASE to replace its original expression in each iteration. The second algorithm picks multiple nodes to shrink simultaneously. This process was formulated as a knapsack problem and a dynamic programming-based approach was proposed to find the optimal solution. Experimental results on a range of benchmarks demonstrated that our approach is not only more effective in area reduction but also much faster than the state-of-the-art approach [6]. In our future work, we will extend the proposed method to handle ALS problem under both the error rate and the error magnitude constraints.

# ACKNOWLEDGMENTS

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

# 8. REFERENCES

[1] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD'13*, pp. 779–786.

[2] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *ISIC'09*, pp. 69–72.

[3] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *ICCAD'13*, pp. 130–137.

[4] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for errortolerant application," in *EDSSC'10*, pp. 1–4.

[5] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSID'11*, pp. 346–351.

[6] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *DATE'13*, pp. 796–801.

[7] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *DATE'10*, pp. 957–960.

[8] ——, "A new circuit simplification method for error tolerant applications," in *DATE'11*, pp. 1–6.

[9] S. Venkataramani *et al.*, "SALSA: Systematic logic synthesis of approximate circuits," in *DAC'12*, 2012, pp. 796–801.

[10] J. Miao, A. Gerstlauer, and M. Orshansky, "Multi-level approximate logic synthesis under general error constraints," in *ICCAD'14*, pp. 504–510.

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

[12] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 6, pp. 1062–1081, 1987.

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

[14] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[15] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE'05*, pp. 412–417.

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

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

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