A Novel Heuristic Search Method for Two-level Approximate Logic Synthesis

Sanbao Su, Chen Zou, Weijiang Kong, Jie Han, *Senior Member, IEEE*, and Weikang Qian, *Member, IEEE*

***Abstract*—Recently, much attention has been paid to approximate computing, a novel design paradigm for error-tolerant applications. It can significantly reduce area, power, and delay of circuits by introducing an acceptable amount of error. In this paper, we propose a new heuristic method for two-level approximate logic synthesis. The problem is to identify an approximate sum-of-product (SOP) expression under a given error rate constraint so that it has the fewest literals. The basic idea of our method is to find an optimal set of input combinations for 0-to-1 output complement (SICC). For this purpose, we first identify all prime SICCs, which are fundamental SICCs in the sense that the optimal SICC is very likely to be a union of a subset of the prime SICCs. Then, we search among all subsets of the prime SICCs the optimal subset, which leads to a final good approximate SOP. We further propose four speed-up techniques. The experiments on benchmarks showed that our method is better than the previous state-of-the-art method and our speed-up techniques are effective. For an error rate threshold of 0.8%, our method can reduce 15.8% literals on average.**

***Index Terms*—Approximate computing, two-level logic synthesis, approximate logic synthesis, literal reduction**

I. Introduction

Approximate computing is an emerging design paradigm [1]–[3]. It targets at error-tolerant applications such as digital signal processing, multimedia, and machine learning. Its basic idea is to deliberately introduce a small amount of error into the circuits. If the error is introduced properly, significant reduction in area, delay, and power consumption [4] can be achieved.

The research on approximate computing circuits can be divided into two sub-areas: manual approximate circuit design and approximate synthesis. The former manually designs approximate circuits for common arithmetic units such as adders [5]–[7] and multipliers [8]–[10]. The latter designs algorithms to automatically synthesize a good approximate version for an arbitrarily given circuit. It is known as *approximate logic synthesis (ALS)*. Several works on ALS proposed techniques to synthesize multi-level circuits, including combinational circuits [11]–[22] and sequential circuits [23]. There are also a few works on ALS targeting at two-level designs [24]–[26]. Given that two-level design usually plays an important role in synthesizing multi-level circuits [27], this work focuses on ALS for two-level designs.

To measure the error of approximate circuits, several error metrics are proposed, such as error rate (ER), worst case error (WCE), mean absolute error (MAE), and Hamming distance (HD) [1], [28]. Among them, two widely used are ER, which is defined as the ratio of input combinations that produce wrong outputs, and WCE, which is defined as the maximal numerical deviation of an incorrect output from a correct one. For this reason, some previous works consider two forms of the ALS problem [24], [25]. One takes ER as the constraint and the other takes WCE as the constraint. It is shown in [25] that the ALS problem under the WCE constraint can be transformed into a Boolean relation minimization problem [29], an extensively studied problem in traditional logic synthesis. However, it is not easy to convert the ALS problem under the ER constraint into a well-solved traditional logic synthesis problem. Thus, we believe this problem requires more intensive study. Hence, it is the target of this work.

In summary, we consider the following fundamental ALS problem here: synthesizing an optimal approximate two-level circuit under an ER constraint. A two-level design is essentially a *sum-of-product (SOP)* representation of a Boolean function. A quality measure of it is the number of literals of the SOP. Thus, our target is to synthesize an approximate SOP with the fewest literals and the ER no larger than a given threshold.

Although there are a few previous works tackling the same problem [24], [26], they can only effectively handle single-output designs. In this work, we propose a novel method to solve the problem. It can effectively handle general multiple-output designs. The fundamental operation that we use to introduce errors is to complement the outputs of some input combinations from 0 to 1. With this, we can form larger cubes that replace some existing ones and hence, reduce the literals. Then, our task becomes identifying an optimal set of input combinations for 0-to-1 output complement (SICC). To solve this problem, we propose the concept of prime SICCs, which are fundamental SICCs in the sense that the optimal SICC is very likely to be a union of some prime SICCs. Our proposed method begins by identifying all prime SICCs. Then, it searches among all subsets of the prime SICCs for the optimal subset, which leads to a final good approximate SOP. To accelerate our method, we further propose four speed-up techniques. The experiments on benchmarks showed that our method can produce approximate SOPs with fewer literals than the state-of-the-art method. It can reduce 15.8% literals on average for an ER of 0.8%.

In summary, the contributions of our work are as follows.

1) We advance the theory of solving the ALS problem under the ER constraint by proposing the concept of prime SICCs. It is fundamental to the problem.
2) Exploiting the concept of prime SICCs, we propose a heuristic search method to identify the approximate SOP with fewer literals under the given ER constraint.
3) We propose four speed-up techniques that can significantly accelerate the basic method.
4) We compare our method to the state-of-the-art method. The experimental results show that our method can achieve much better results, especially for multiple-output circuits.

The rest of the paper is organized as follows. Section II discusses the related works. Section III introduces some preliminaries, including the terminology used in this work. Section IV

Sanbao Su, Weijiang Kong, and Weikang Qian are with the University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China; email: gawaine@sjtu.edu.cn, 5143709071@sjtu.edu.cn, qianwk@sjtu.edu.cn.

Chen Zou is with the Department of Computer Science, the University of Chicago, Chicago, IL, USA; email: chenzou@uchicago.edu.

Jie Han is with the Department of Electrical and Computer Engineering, the University of Alberta, Edmonton, AB, Canada; email: jhan8@ualberta.ca.

presents the basic heuristic search method. Section V presents the speed-up techniques. Section VI shows the experimental results. Finally, Section VII concludes the paper.

II. Related Works

In [24], Shin and Gupta designed a two-level ALS method to identify an approximate SOP with fewer literals. The problem they considered is the same as ours. Their basic idea is to complement the output values of some minterms from 0 to 1 to decrease literals in the final SOP under the ER constraint. Their method first obtains a set of candidate minterms for complement. Then, it traverses all combinations of candidate minterms for complement and identifies the one with the largest reduction in estimated literals.

In [26], Zou *et al.* proposed a dynamic programming-based solution for the same problem. They introduced a quality metric called performance vector (PV) for a set of cubes. PV has a weak correlation to the literal reduction when the set of cubes is added into the original SOP. Their basic idea is to identify the set of cubes with the largest PV under the ER constraint and add the set into the SOP. However, PV is not always proportional to the literal reduction.

In [25], Miao *et al.* proposed a two-phase greedy method for two-level ALS. Their work considers both the ER and the WCE constraints. Their method first ignores the ER constraint and minimizes the SOP under the WCE constraint only. This problem is isomorphic to the Boolean relation minimization problem [29] and can be solved using an existing heuristic algorithm [30]. Then, their method iteratively refines the solution to reach a solution also satisfying the ER constraint. However, this method cannot achieve the optimal result for ER constraint since it prioritizes on the WCE constraint.

A common drawback of the previous methods is that they are mainly designed for single-output circuits. For a multiple-output circuit, they separate the circuit into several single-output ones and then handle them individually. Thus, they are not optimal for multiple-output circuits. Our method can handle multiple-output circuits better.

III. Preliminaries

A. Terminology of Two-level Logic Synthesis

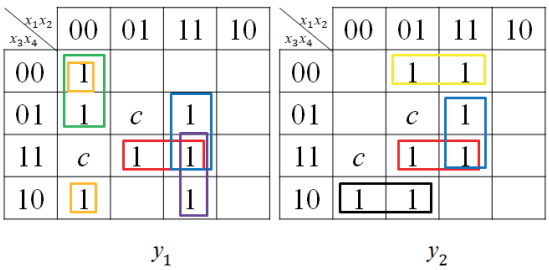


Figure 1: The Karnaugh map of a 4-input 2-output function. The 1s give the on-set of the original function. A solid-line rectangle corresponds to a cube in the original SOP expression. The c's denote some erroneous input combinations (EICs), which will be used in some examples in the paper.

In this section, we introduce some definitions related to two-level logic synthesis, which are used in this paper. We assume that the Boolean function is completely specified without any don't cares. We will study how to extend the proposed method to handle functions with don't cares in the future. Unless otherwise specified, these definitions are defined over general m-input n-output Boolean functions.

1) **Variable**: A symbol used to represent an input or an output signal. In what follows, we use x_i and y_i to represent the i-th input and output signals, respectively.
2) **Input literal**: An input variable with or without negation. For example, both x_1 and $\overline{x_1}$ are input literals.
3) **Output literal**: An output variable without negation.
4) **Literal**: An input literal or an output literal.
5) **Input cube**: A product of input literals, where for each input variable x, x and $\overline{x}$ do not appear simultaneously. For example, the product $x_1\overline{x_2}$ is an input cube. An input cube defines a single-output function, which is 1 for all input combinations that let the product be 1, and 0 for the remaining input combinations.
6) **Cube**: A product of an input cube and k output literals, where $k \geq 1$. Here, we essentially call a product of literals *cube*. Note that to efficiently handle multiple-output functions, our definition of cubes also includes the output signals. A cube is a special multiple-output function: if the output y_i is not in the cube, then the i-th output of the function is always 0; otherwise, the i-th output is just the input cube. For example, assume $m = 4$ and $n = 3$. Then, a cube $x_1\overline{x_2}y_1y_2$ is a 3-output function such that the first and the second outputs are equal to $x_1\overline{x_2}$ and the third output is a constant 0. Given this definition of a cube, we can represent a multiple-output Boolean function as a sum of cubes. For example, the multiple-output Boolean function of Fig. 1 is

$$\begin{aligned} f = {} & \overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1 + \overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1 + x_1x_2x_4y_1y_2 \\ & + x_2x_3x_4y_1y_2 + x_1x_2x_3y_1 \\ & + x_2\overline{x_3}\ \overline{x_4}y_2 + \overline{x_1}x_3\overline{x_4}y_2. \end{aligned} \tag{1}$$

   A widely-used quality measure for an SOP is its number of literals. For example, the number of literals of the SOP in Eq. (1) is 30.
7) **Input minterm**: A special input cube where each input variable appears exactly once, in either complemented or uncomplemented form. For example, for $m = 4$, $x_1x_2\overline{x_3}x_4$ is an input minterm. Since an input minterm uniquely corresponds to an input combination that lets the minterm be 1, we will use the terms input minterm and input combination interchangeably.
8) **Minterm**: A special cube as the product of an input minterm and *exactly one* output variable. For example, for $m = 4$, the cubes $x_1x_2\overline{x_3}x_4y_1$ and $x_1\overline{x_2}\ \overline{x_3}x_4y_2$ are minterms. Given m inputs and n outputs, the total number of minterms is $2^m n$. Given a function f, the entire set of minterms can be partitioned into 2 subsets, *on-set* and *off-set* of f. For an arbitrary minterm, suppose its output variable is y_i. It is in the on-set/off-set of f if for the input combination that lets the input minterm be 1, the i-th output of f is 1/0. We say the minterm is *covered* by f if it is in the on-set of f.
9) **Size of a cube**: the number of minterms covered by the cube. For example, assume $m = 4$ and $n = 3$. Then, the size of the cube $x_1\overline{x_2}y_1y_2$ is 8.
10) **Erroneous input combination (EIC)** of an approximate function: An input combination for which the approximate function is different from the original function in at least one output. Complementing some outputs of an input combination of the original function makes that input combination an EIC of the approximate function.
11) **Covered-once minterm (COM)** of an SOP: a minterm that is only covered by one cube in the SOP expression. For example, for the SOP in Eq. (1), which corresponds

to the set of solid-line rectangles in Fig. 1, the minterm $\overline{x_1}x_2x_3x_4y_1$ is a COM, since it is only covered by the cube $x_2x_3x_4y_1y_2$, as shown in Fig. 1. If a COM is covered by a cube c in the SOP, we will also refer to it as a covered-once minterm (COM) of the cube c.

B. Hasse Diagram on Cubes

Hasse diagram is a directed acyclic graph used to represent a finite partially ordered set [31]. Here, we introduce a Hasse diagram defined on cubes of m inputs and n outputs, which will be used in our work. Fig. 2 is a Hasse diagram on cubes of 2 inputs and 2 outputs. Each node in the diagram corresponds to a cube. The diagram contains all possible cubes of m inputs and n outputs and organizes them into multiple levels. There is only one node at the first level. The node is a cube including all output variables, but no input variables. It corresponds to an m-input n-output Boolean function that covers all possible minterms. The last level has $2^m n$ nodes, each corresponding to one of the minterms. Each cube at a level other than the last level is connected to multiple cubes at the next level, forming a parent-child relation. A child cube is formed from a parent cube by adding an input literal of a non-existing input variable or removing an output literal of the parent cube. Therefore, a child cube only has one different literal than its parent cube. For example, as shown in Fig. 2, the child cubes of the cube $x_1y_1y_2$ are $x_1x_2y_1y_2$, $x_1\overline{x_2}y_1y_2$, x_1y_1, and x_1y_2. A parent cube covers all minterms of a child cube of it. It is easy to see that a Hasse diagram defined on cubes of m inputs and n outputs has $3^m(2^n-1)$ nodes.

IV. Methodology

As stated in Section I, the problem we consider is to identify an approximate SOP expression with the fewest literals for a given original expression F and an ER threshold T_E. The given SOP expression is an optimized one processed by Espresso [32], an off-the-shelf two-level logic synthesis tool. By the definition of *ER* in Section I, the ER threshold T_E means that the number of input combinations that produce wrong outputs should be no larger than $T_E \cdot 2^m$, where m is the number of inputs of F. By the definition of *EIC* in Section III-A, an input combination that produces wrong outputs is just an EIC. Thus, the ER constraint means that the number of EICs of the approximate expression must be no larger than $T_E \cdot 2^m$. Note that this bound, $T_E \cdot 2^m$, is independent of the number of outputs of the function. We define $e = T_E \cdot 2^m$ and call it *number of errors (NoE) threshold*. Next, we first present the basic strategy and then elaborate our solution.

A. Basic Strategy

There are two ways to introduce errors: flipping one output of an input combination from 0 to 1 (hereafter referred to as *0-to-1 complement*) and flipping that from 1 to 0 (hereafter referred to as *1-to-0 complement*). In [24], an experiment showed that 0-to-1 complements are typically more beneficial than 1-to-0 complements. The reason is that a 1-to-0 complement can only remove at most one existing cube due to the removal of a minterm from the on-set. In contrast, a 0-to-1 complement adds a new minterm to the on-set. Thus, it may form new and larger cubes to make some existing cubes redundant.

Example 1
An example for 0-to-1 complement is shown in Fig. 3. Fig. 3a shows the Karnaugh map of the original function, which is single-output. If we complement the output of the input minterm $\overline{x_1}x_2x_3x_4$ in the original function from 0 to 1, we derive an approximate function shown in Fig. 3b. In this case, a new and larger cube $x_2x_4y_1$ is formed, which makes two original cubes $x_1x_2x_4y_1$ and $x_2\overline{x_3}x_4y_1$ redundant. Consequently, the number of literals is reduced by 5. □

In summary, 0-to-1 complement has the potential of removing more literals than 1-to-0 complement. Thus, in this work, we only consider 0-to-1 complement as the changes to the original function. In what follows, when we say complementing an input minterm/combination, we mean changing some outputs of that input minterm/combination from 0 to 1.

In order to produce an approximate function, we only need to find a set of input combinations, each with some outputs in the original function evaluated to 0, and complement some of these outputs from 0 to 1 for each input combination. We call such a set *a set of input combinations for 0-to-1 output complement (SICC)*. First, we have the following claim.

Theorem 1
If we produce an approximate function by complementing each input combination in an SICC, then each element in the SICC is an EIC for the approximate function. □

Proof: Consider an input combination v in the SICC. By the way to construct the approximate function, it is different from the original function in at least one output for v. Thus, by the definition of EIC in Section III-A, v is an EIC. □

In order to find an optimal approximate SOP under the ER constraint, our task is to find an SICC satisfying the following properties. First, by Theorem 1, each element in the SICC is an EIC. Since we require the number of EICs to be no larger than e, therefore, the size of the SICC should be no larger than e. Second, complementing the input combinations in the SICC should cause the largest literal reduction. This is because we produce an approximate SOP by complementing the input combinations in the SICC and we want to synthesize an optimal approximate SOP with the fewest literals.

B. Overview of Our Heuristic Method

A straightforward way to find the optimal SICC is to enumerate all possible SICCs and choose the best. However, due to the existence of an exponential number of SICCs, this method is impractical. Instead, we propose a heuristic search method to find a good solution. The basic idea is to first identify a set of *prime SICCs*. They are fundamental SICCs in the sense that any promising SICC can be built as a union of some prime SICCs. Once the set of prime SICCs is identified, we will find an optimal subset of it so that the union of the SICCs in the subset gives the maximal literal reduction.

The entire flow is shown in Alg. 1. The first step is to build a set of *SICC-cube trees* (see Line 3). Their roots give the prime SICCs. We will describe the details of this step in Section IV-C. The second step is to update the SICC-cube trees to build *augmented SICC-cube trees* (see Line 4). The root of an augmented SICC-cube tree is the same prime SICC as the root of the corresponding SICC-cube tree. We will describe the details of this step in Section IV-D. The third step is to synthesize the final optimal approximate SOP expression by identifying an optimal subset of the prime SICCs (see Line 5). It is based on the set of augmented SICC-cube trees obtained

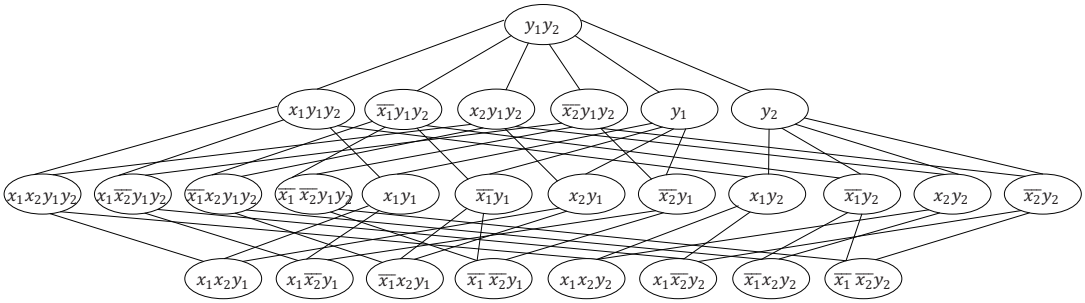
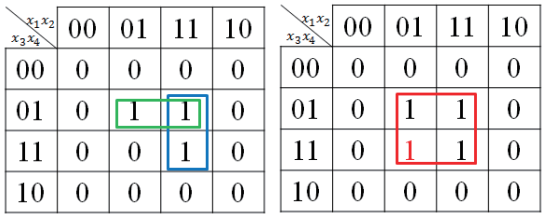


Figure 2: The Hasse diagram on cubes of 2 inputs and 2 outputs.



(a) (b)

Figure 3: An example of 0-to-1 complement. (a) The Karnaugh map of the original function $f = x_1x_2x_4y_1 + x_2\overline{x_3}x_4y_1$; (b) The Karnaugh map of the approximate function $f = x_2x_4y_1$.

in the second step. We will describe the details of this step in Section IV-E.

Algorithm 1: The proposed heuristic search method.

Input : a simplified SOP expression F and an error rate threshold T_E.
Output : an approximate SOP expression F'.

```
m ← number of inputs of F;
NoE threshold e ← T_E · 2^m;
the set of SICC-cube trees S ← SICCCubeTree (F, e);
the set of augmented SICC-cube trees T ← AugTree (S);
return SynthOpt (T, F, e);
```

C. First Step: Building SICC-Cube Trees

The first step is to build a set of *SICC-cube trees* for the given SOP F and the NoE threshold e. To introduce the SICC-cube trees, we first give the following definition.

Definition 1
An **erroneous input combination (EIC) of a cube** *is an input combination for which there exists an output for which the cube evaluates to 1, while the original function evaluates to 0.* □

For example, for the Boolean function shown in Fig. 1, input combination $\overline{x_1}\ \overline{x_2}x_3x_4$ (denoted by two c's in the figure) is an EIC of cube $\overline{x_1}\ \overline{x_2}y_1$, because for this input combination and output y_1, the cube evaluates to 1, while the original function evaluates to 0.

Now, we introduce the definition of an SICC-cube tree.

Definition 2
An **SICC-cube tree** *is a two-level tree. Its root is an SICC of size no larger than the NoE threshold e. Its second level is composed of cubes satisfying that*

1) *the set of EICs of the cube equals the SICC at the root;*
2) *adding the cube into the original SOP expression removes at least one existing cube and the total number of literals in the SOP does not increase.*

The root of an SICC-cube tree is called a **prime SICC**. □

Example 2
Fig. 4a shows an example of an SICC-cube tree for the SOP indicated in Fig. 1 (see Eq. (1)) and the NoE threshold $e = 2$. The root is an SICC of a single input minterm $\overline{x_1}\ \overline{x_2}x_3x_4$. Its second level is composed of three cubes. For each cube, its set of EICs equals the SICC at the root. Furthermore, adding the cube into the given SOP removes at least one existing cube and the total number of literals in the SOP does not increase. For example, the cube $\overline{x_1}\ \overline{x_2}x_4y_1$ has its set of EICs same as the SICC at the root, i.e., the set $\{\overline{x_1}\ \overline{x_2}x_3x_4\}$. By adding it into the original SOP, we can remove the original cube $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ and the total number of literals does not change.

By Definition 2, the set $\{\overline{x_1}\ \overline{x_2}x_3x_4\}$, which is the root of the SICC-cube tree, is a prime SICC. □

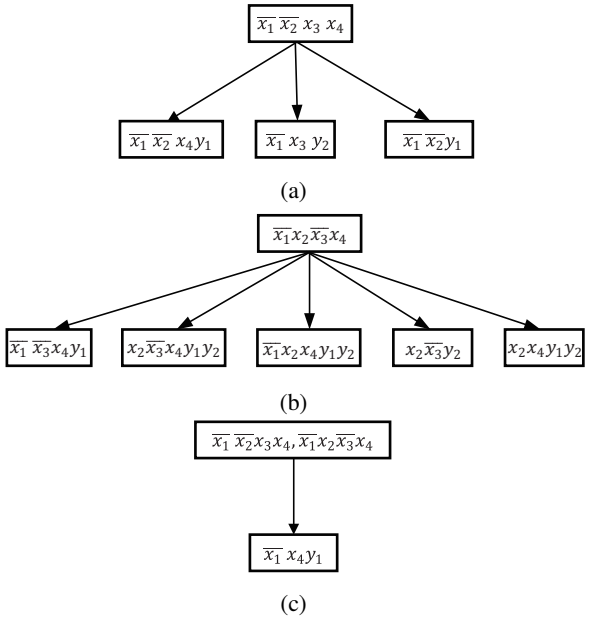


Figure 4: Three SICC-cube trees for the SOP indicated in Fig. 1 (see Eq. (1)) and the NoE threshold $e = 2$. Their root SICCs are: (a) $\{\overline{x_1}\ \overline{x_2}x_3x_4\}$; (b) $\{\overline{x_1}x_2\overline{x_3}x_4\}$; and (c) $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$.

It should be noted that by Definition 2, a prime SICC is an SICC of size no larger than the NoE threshold e. However, the reverse is not true. This is because for an SICC of size no larger than e, we may not find any cube such that the set of

EICs of the cube equals the SICC. Thus, we cannot construct an SICC-cube tree with the SICC as the root. Therefore, the SICC may not be a prime SICC.

The concept of prime SICC is important because the SICC that gives the final optimal approximate SOP expression is *very likely* to be formed as a union of some prime SICCs. We argue the reason as follows. The final optimal approximate SOP expression is constructed by adding some new cubes into the original SOP and removing some existing cubes. The number of EICs of each new cube should be no more than the NoE threshold e. Furthermore, although not guaranteed, it is *very likely* that each new cube added should satisfy that after it is added into the original SOP and the redundant cubes are removed, the literal count does not increase. By Definition 2, such a cube belongs to an SICC-cube tree. Correspondingly, the set of EICs introduced due to the addition of the new cube is the prime SICC at the root of the SICC-cube tree. Therefore, the final optimal SICC is *very likely* to be formed as a union of some prime SICCs.

Given the important relation between prime SICCs and the final optimal SICC, in this step, we try to identify all prime SICCs by finding the set of SICC-cube trees. To directly identify the prime SICCs at the roots of the SICC-cube trees could be challenging. Instead, we try to identify the possible cubes in the second level of the SICC-cube trees. These cubes should satisfy the following criteria.

1) The number of EICs of the cube should be at least 1 and at most e. This is because the cube belongs to the SICC-cube tree of which the SICC at the root equals the set of EICs of the cube. Since the SICC at the root is nonempty and its size is no more than e, the number of EICs of the cube should be at least 1 and at most e. For the Boolean function shown in Fig. 1 and $e = 2$, the cube $x_2x_4y_1y_2$ can be considered, since it contains one EIC $\overline{x_1}x_2\overline{x_3}x_4$. However, the cube x_4y_1 cannot be considered, because it has 4 EICs $\overline{x_1}x_2\overline{x_3}x_4$, $x_1\overline{x_2}\ \overline{x_3}x_4$, $\overline{x_1}\ \overline{x_2}x_3x_4$, and $x_1\overline{x_2}x_3x_4$.
2) Adding the cube to the original SOP should remove at least one existing cube in the original expression. Here, for the simplicity of processing, we do not consider cube expansion induced by the added cube. Thus, if the added cube could remove an existing cube c, it should cover all of c's COMs. Consider the Boolean function in Fig. 1. The cube $\overline{x_1}\ \overline{x_3}x_4y_1$ can remove the existing cube $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ (the cube in the green rectangle) by covering its only COM $\overline{x_1}\ \overline{x_2}\ \overline{x_3}x_4y_1$. This criterion is necessary because otherwise, adding the cube may increase the literals.
3) The literal count of the cube should be no larger than the sum of the literal counts of the existing cubes removed by it. This means that adding this cube and removing the corresponding existing cubes do not increase the literals.

The flow of the first step is shown in Alg. 2. To identify the candidate cubes, we scan all cubes in the Hasse Diagram in a breadth-first order (see Line 2). At the same time, we maintain a growing set of SICC-cube trees. For each cube c we visit, we judge whether it satisfies the above criteria (see Lines 3–5). If it does, we identify its set of EICs R (see Line 6). Then, we search the set of the SICC-cube trees. If there exists an SICC-cube tree with the root SICC same as R, we add the cube to the second level of that tree (see Lines 7–8). Otherwise, we create a new SICC-cube tree with the root SICC as set R and the second level consisting of only cube c (see Lines 9–11).

For the convenience of later steps, for each SICC-cube tree, we order the cubes in the second level from left to right in increasing size of a cube. If there is a tie, we further order them in increasing number of existing cubes that can be removed by the cube.

Algorithm 2: The function `SICCCubeTree`(F, e) for building the set of SICC-cube trees.

Input : a simplified SOP expression F and an NoE threshold e.
Output : the set of SICC-cube trees S for the given F and e.

```
1  the set of SICC-cube trees S ← ∅;
2  for each cube c in the Hasse Diagram in the breadth-first traversal order do
3      if the number of EICs of c is at least 1 and at most e then
4          find all cubes in F such that all of their COMs are covered by c, and compute the sum of the literals L of these cubes;
5          if L is no less than the literal count of c then
6              obtain the set R of EICs of c;
7              if there is an SICC-cube tree of root R in set S then
8                  add c to the second level of the tree of root R;
9              else
10                 build an SICC-cube tree with the root as R and the second level only with cube c;
11                 add the new tree into set S;
12 return S;
```

Example 3
Figs. 4a and 4b are two SICC-cube trees for the SOP indicated in Fig. 1 and the NoE threshold $e = 2$. In Fig. 4a, the cubes $\overline{x_1}x_3y_2$ and $\overline{x_1}\ \overline{x_2}y_1$ have the same size, but $\overline{x_1}\ \overline{x_2}y_1$ can remove more existing cubes than $\overline{x_1}x_3y_2$. Specifically, $\overline{x_1}\ \overline{x_2}y_1$ can remove two existing cubes $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ (the cube in the green rectangle) and $\overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1$ (the cube in the orange rectangle), while $\overline{x_1}x_3y_2$ can only remove one existing cube $\overline{x_1}x_3\overline{x_4}y_2$ (the cube in the black rectangle). Therefore, $\overline{x_1}\ \overline{x_2}y_1$ is on the right of $\overline{x_1}x_3y_2$. In Fig. 4b, the cube $x_2x_4y_1y_2$ has a larger size than the other cubes, so it is at the rightmost side. □

When we build the set of SICC-cube trees, we also avoid unnecessary cube checking in the Hasse diagram. For example, when we find a cube that has no EICs, we will ignore its descendant cubes in the Hasse diagram, because each descendant cannot have any EICs and hence, does not satisfy the first criterion above.

D. Second Step: Building Augmented SICC-Cube Trees

The second step is to update the SICC-cube trees obtained in the first step. Each SICC-cube tree only contains cubes such that their sets of EICs equal the prime SICC at the root. However, if the prime SICC is selected to form the final optimal SICC, the cubes that can be added into the original SOP are not just limited to those cubes. The cubes such that their sets of EICs are the *subsets* of the SICC at the root can also be added into the original SOP. For example, consider the SICC-cube tree shown in Fig. 4c. If its root SICC is selected, then the cubes that can be added into the original SOP are not just limited to its only cube at the second level, i.e., $\overline{x_1}x_4y_1$; they also include the cubes such that their sets of EICs are either $\{\overline{x_1}\ \overline{x_2}x_3x_4\}$ or $\{\overline{x_1}x_2\overline{x_3}x_4\}$, which is a subset of the SICC $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$.

Thus, in this step, we augment the second level of each SICC-cube tree t by merging all cubes in the SICC-cube trees such that their SICCs are subsets of the SICC of the tree t. We call such an SICC-cube tree an *augmented SICC-cube*

tree. The cubes in the second level of an augmented SICC-cube tree are also ordered according to the rules described in Section IV-C. For example, for the SICC-cube tree in Fig. 4c, we add the cubes in the SICC-cube trees shown in Figs. 4a and 4b into its second level. The result is the augmented SICC-cube tree of the SICC $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$. It is shown in Fig. 5. We denote this step as `AugTree(`S`)`, where S is a set of SICC-cube trees.

E. Third Step: Identifying Optimal Subset of Prime SICCs

From the first step, we have identified all the prime SICCs. In this step, we identify the optimal subset of the prime SICCs that causes the maximal literal reduction.

Algorithm 3: The function `SynthOpt`(T, F, e) for synthesizing a good approximate SOP expression.

Input : a set T of augmented SICC-cube trees, a simplified SOP expression F, and an NoE threshold e.
Output : an approximate SOP expression F'.

1 the optimal result array $R \leftarrow$ `Search` (T, F, e);
2 $bestNoE \leftarrow$ the index of the entry in R that gives the largest literal reduction;
3 construct the approximate SOP expression F' from the SICC of the tree $R[bestNoE].t$;
4 **return** F' *simplified by Espresso;*

The flow of this step is shown in Alg. 3. The major work is done by the function `Search` (T, F, e) (see Line 1). It enumerates all subsets of the prime SICCs and returns the optimal result array R. Specifically, the function enumerates all *combined SICC-cube trees*. A combined SICC-cube tree is a combination of some augmented SICC-cube trees: its root is the union of the prime SICCs of these augmented trees; its second level is the union of all cubes in the second levels of these augmented trees. The cubes in the second level of a combined SICC-cube tree are ordered according to the rules described in Section IV-C. After the function `Search` ends, it returns an optimal result array R of size $(e+1)$, where e is the given NoE threshold. The i-th $(0 \leq i \leq e)$ entry of the array R, $R[i]$, involves two components, $R[i].mlr$ and $R[i].t$. $R[i].mlr$ records the maximal literal reduction among all prime SICC unions with size equal to i, while $R[i].t$ records the combined SICC-cube tree that gives that specific maximal literal reduction. Note that the array includes the entry $R[0]$, since it corresponds to the special case where no error is introduced to the original function, which is also a candidate for the final best solution. An important subroutine in the function `Search` is to estimate the literal reduction for a union of some prime SICCs. We will discuss this subroutine in Section IV-F.

After the array R is constructed, Line 2 identifies the index of the entry in R that gives the largest literal reduction among all entries in R. Note that given NoE threshold e, it is not necessary that the maximal literal reduction occurs for an SICC with size exactly equal to e. It is possible that it occurs for an SICC with size smaller than e. This explains why we build array R and scan it to find the entry of the maximal literal reduction. Suppose the index we have identified is $bestNoE$. Then, Line 3 constructs the approximate SOP expression F' from the combined SICC-cube tree $R[bestNoE].t$. The construction of the approximate expression is achieved as a byproduct of calling the subroutine for estimating the literal reduction, which will be described in Section IV-F. Finally, Line 4 simplifies F' by Espresso and returns it.

F. Estimating the Literal Reduction

One important procedure is to estimate the literal reduction of the union of a subset of the prime SICCs. It takes a combined SICC-cube tree as an input and estimates the literal reduction of the SICC at the root of that tree. We can leverage Espresso [32] to do this. Specifically, we identify the outputs of the input combinations in the SICC that are 0 and set these outputs as don't cares. Then, we apply Espresso to simplify this modified Boolean function. The final literal reduction is the difference between the literal count of the original expression and that of the simplified expression. However, since each subset of the prime SICCs requires one call of Espresso, the total runtime could be large.

In this section, we describe a more efficient procedure to estimate the literal reduction. As a byproduct, this procedure also tells how to construct the approximate function that gives the estimated literal reduction. Note that at Line 3 of Alg. 3, we build the approximate SOP expression for the identified optimal SICC. This is achieved as a byproduct of applying this procedure. In what follows, for simplicity, we will refer to a combined SICC-cube tree as an SICC-cube tree.

To obtain the exact number of decreased literals is computationally intractable. Instead, we propose a heuristic method that gives an estimation. The overall flow of this heuristic method is shown in Alg. 4. It has the following three steps.

1) Identifying the maximal set M_1 of the existing cubes in the original SOP that can be removed by adding all cubes in the second level of the SICC-cube tree t. It corresponds to Lines 3–9 in Alg. 4. The reason for this step is because we want to remove as many existing cubes as possible to achieve the maximal literal reduction.
2) Identifying the minimal set M_2 of the cubes in the second level of the tree t so that adding them into the original SOP can remove the cubes in set M_1. It corresponds to Lines 10–13 in Alg. 4. The reason for this step is because in order to remove the cubes in set M_1, we need to add cubes. However, each added cube increases the literals. Thus, we want to minimize the number of added cubes.
3) Calculating the literal reduction due to the addition of the cubes in set M_2 and the removal of the cubes in set M_1. It corresponds to Line 14 in Alg. 4.

We elaborate these three steps in the following sub-sections.

1) Identifying the Maximal Set of Existing Cubes that Can Be Removed: This step is shown in Lines 3–9 in Alg. 4. To identify the maximal set M_1 of the existing cubes in the original SOP that can be removed, we construct the *SICC-cube graph* from the SICC-cube tree.

Fig. 6 shows the SICC-cube graph of the SICC $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$ constructed from the SICC-cube tree shown in Fig. 5. The first and the second levels of the SICC-cube graph are the same as those of the SICC-cube tree. Its third level is composed of all covered-once minterms (COMs) of the cubes in the fourth level. Its fourth level is composed of all cubes in the original SOP that can be removed by adding all cubes in the second level. Thus, its fourth level gives the maximal set M_1. The edge between a COM v in the third level and a cube c in the fourth level means that v is a COM of cube c. The edge between a COM v in the third level and a cube c in the second level means that cube c covers COM v. To show the graph clearly, the edges starting from the same cube in the second level are drawn with the same color and style.

Next, we describe how we build the SICC-cube graph in detail. We first identify the *covered on-set* (see Line 3), which

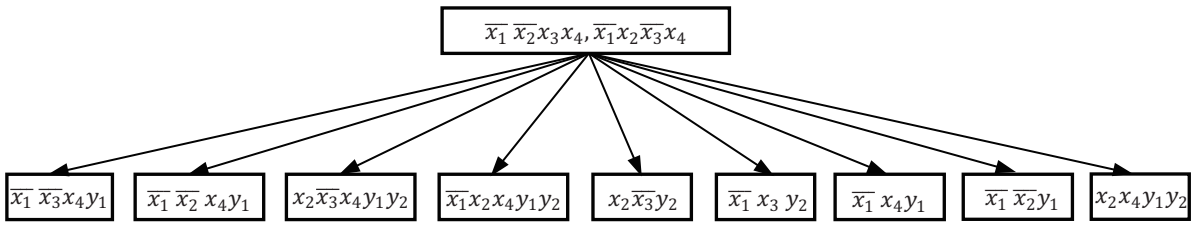


Figure 5: The augmented SICC-cube tree of the prime SICC $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$ for the SOP indicated in Fig. 1 and the NoE threshold $e = 2$.

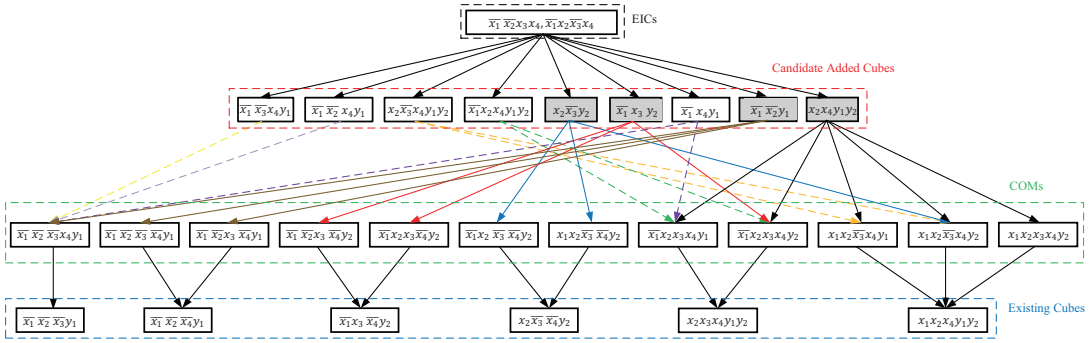


Figure 6: The SICC-cube graph of the SICC $\{\overline{x_1}\ \overline{x_2}x_3x_4, \overline{x_1}x_2\overline{x_3}x_4\}$ for the SOP indicated in Fig. 1 and the NoE threshold $e = 2$.

Algorithm 4: The function `LiteralReduct`(F, t) for estimating the literal reduction of a combined SICC-cube tree.

Input : the original simplified SOP expression F and a combined SICC-cube tree t.
Output : the estimated literal reduction dl.

1 the maximal existing cube set $M_1 \leftarrow \varnothing$;
2 the minimal cube set $M_2 \leftarrow \varnothing$;
3 covered on-set $S_o \leftarrow$ all on-set minterms of F that are covered by the cubes in the second level of tree t;
4 **for** *each minterm* v *in* S_o **do**
5 **for** *each cube* c *of* F *covering* v **do**
6 **if** *all COMs of* c *are in* S_o **then**
7 **if** c *has not been added into the fourth level of* t **then**
8 add all COMs of c into the third level of t, add c into the fourth level of t, and add c into M_1;
9 remove c from F and update COMs of the other cubes of F;
10 $S_c \leftarrow$ all COMs in the third level of t;
11 **for** *each cube* c *in the second level of* t *from right to left* **do**
12 **if** c *covers some COMs in* S_c **then**
13 add c into M_2 and remove the COMs covered by c from S_c;
14 the estimated literal reduction $dl \leftarrow$ the sum of the literal counts of the cubes in M_1 minus that of the cubes in M_2;
15 **return** dl;

is composed of all on-set minterms of the original SOP that are covered by the cubes in the second level of the SICC-cube tree. For example, Fig. 7 duplicates the Karnaugh map shown in Fig. 1. The shaded 1s in Fig. 7 form the covered on-set of the SICC-cube tree shown in Fig. 5.

Then, we iterate over all minterms in the covered on-set (see Line 4). For each minterm v in the covered on-set, we further iterate over all cubes in the original SOP that cover v (see Line 5). For each cube c, we check whether it can be removed due to the addition of the cubes in the second level of the tree. If all COMs of c are in the covered on-set (see

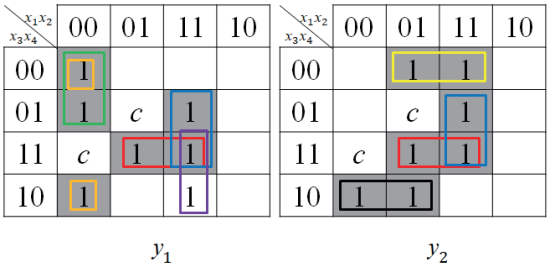


Figure 7: The same Karnaugh map as that shown in Fig. 1. A shaded 1 corresponds to an on-set minterm of the original SOP that is covered by the cubes in the second level of the SICC-cube tree in Fig. 5.

Line 6), c can be removed. In this case, if c has not been added into the SICC-cube graph, we add its COMs and the cube c into the third and the fourth level of the SICC-cube graph, respectively (see Lines 7–8). For each COM added into the third level and each cube in the second level that covers the COM, we draw an edge from the cube to the COM. For each COM added into the third level, we also draw an edge from it to the cube c added in the fourth level.

Example 4
Consider the SOP shown in Fig. 7 and the SICC-cube tree shown in Fig. 5. We first consider the (shaded) minterm $\overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4}y_1$ *in the covered on-set. We can identify two existing cubes,* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ *(the cube in the green rectangle) and* $\overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1$ *(the cube in the orange rectangle) covering that minterm. We first check cube* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$*. It has only one COM, i.e.,* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}x_4y_1$*, and the COM is shaded, which means that the COM is in the covered on-set. Thus, the cube can be removed due to the addition of the cubes in the second level of the SICC-cube tree. Therefore, we add the COM* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}x_4y_1$ *into the third level and the cube* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ *into the fourth level of the SICC-cube graph. For each cube in the second level covering the COM, we draw an edge from the cube to the COM. We also draw an edge from the COM to the cube* $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ *in the fourth level. This is shown in Fig. 6.* □

Each time a cube of the original SOP that can be removed is identified, it is temporarily removed from the original SOP and the COMs of the other cubes of the original SOP are updated immediately (see Line 9). Then, the next iteration begins.

Example 5
Continue from Example 4. We have just identified that cube $\overline{x_1}\ \overline{x_2}\ \overline{x_3}y_1$ can be removed. Then, we temporarily remove the cube from the original SOP. As a result, the set of COMs of cube $\overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1$ should be updated: originally, minterm $\overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4}y_1$ is not a COM of the cube, but now it is. Continuing the overall procedure, we next check cube $\overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1$, since it is the next cube in the original SOP that covers minterm $\overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4}y_1$. Since its COM set contains two COMs and both are in the covered on-set, the cube can be removed. Thus, we also add the two COMs and cube $\overline{x_1}\ \overline{x_2}\ \overline{x_4}y_1$ into the third and fourth level of the SICC-cube graph, respectively. The edges are connected based on the rules mentioned above. We continue the process. Finally, the entire SICC-cube graph is constructed, as shown in Fig. 6. Its fourth level includes all cubes in the original SOP that can be removed by adding the cubes in the second level to the original SOP. Compared to the original SOP shown in Eq. (1), all cubes in the original SOP except cube $x_1x_2x_3y_1$ are in the fourth level, since these cubes can be removed by adding the cubes in the second level to the original SOP. Cube $x_1x_2x_3y_1$ is not in the fourth level. This is because its COM $x_1x_2x_3\overline{x_4}y_1$ is not covered by any cubes in the second level. Thus, adding the cubes in the second level to the original SOP cannot remove cube $x_1x_2x_3y_1$. The third level are the COMs of those cubes in the fourth level. □

It should be noted that the order in which we check the minterms in the covered on-set will influence the graph and hence, the estimated literal reduction. For simplicity, we stick to the order of these minterms in the on-set of the original function.

2) Identifying the Minimal Set of Cubes to Add: In this step, we identify the minimal set of cubes in the second level of the SICC-cube tree that can remove all cubes in the maximal set M_1, which correspond to the cubes in the fourth level of the SICC-cube graph. This step corresponds to Lines 10–13 in Alg. 4. It is done with the help of the SICC-cube graph. First, note that for a cube c in the fourth level of the graph, if a set of cubes in the second level covers all COMs in the third level that connect to c, adding this set of cubes can remove cube c. The following shows an example.

Example 6
Consider cube $x_1x_2x_4y_1y_2$ in the fourth level of the SICC-cube graph shown in Fig. 6. By the graph, its set of COMs is $\{x_1x_2\overline{x_3}x_4y_1, x_1x_2\overline{x_3}x_4y_2, x_1x_2x_3x_4y_2\}$. Also, from the graph, we can see that cube set $\{x_2x_4y_1y_2\}$ covers that set of COMs. Thus, adding cube set $\{x_2x_4y_1y_2\}$ can remove cube $x_1x_2x_4y_1y_2$. □

In order to identify the minimal set of cubes that can remove all cubes in the fourth level, we only need to identify the minimal set that covers all COMs in the third level. This problem can be solved exactly by formulating it as an integer linear programming (ILP) problem. Here, we propose a quick heuristic method. We initialize a set S_c, which consists of all COMs in the third level (see Line 10). We check each cube in the second level from right to left (see Line 11). If the cube under check covers some COMs in set S_c, we add the cube to set M_2 and remove all COMs covered by the cube from set S_c (see Lines 12–13). Otherwise, we do nothing and check the next cube. Note that the check order is from right to left. This is because by our ordering rule, a cube on the right has a larger size and could remove more existing cubes than a cube on the left. Thus, a cube on the right is likely to cover more COMs in the third level than a cube on the left. Below shows an example of how our procedure obtains the minimal set M_2.

Example 7
Consider the SICC-cube graph shown in Fig. 6. The initial set S_c consists of all COMs in the third level of the graph. By the proposed procedure, we first check the rightmost cube in the second level of the graph, i.e., cube $x_2x_4y_1y_2$. Since the cube covers some COMs in S_c, it is added into set M_2. The COMs covered by the cube are removed from set S_c. In this case, the rightmost 5 COMs in the third level are removed. Next, we check cube $\overline{x_1}\ \overline{x_2}y_1$. Since it covers the leftmost 3 COMs in the third level, which are in the current set S_c, it is added into set M_2. The leftmost 3 COMs in the third level are then removed from set S_c. Consequently, S_c only contains the 4th, 5th, 6th, and 7th COMs in the third level. Next, we check cube $\overline{x_1}\ x_4y_1$. However, it does not cover any COMs in set S_c. Thus, we do nothing and check the next cube. After all cubes in the second level are checked, we obtain set M_2 as $\{x_2\overline{x_3}y_2,\ \overline{x_1}x_3y_2,\ \overline{x_1}\ \overline{x_2}y_1,\ x_2x_4y_1y_2\}$. The cubes in set M_2 are those shaded rectangles in the second level of the graph in Fig. 6. They are also marked by the dashed-line rectangles in Fig. 8, which duplicates the Karnaugh map shown in Fig. 1. The COMs in the third level are marked in red in Fig. 8. We can see that all COMs are covered by the set of cubes M_2 we choose. Thus, this cube set can remove all existing cubes in the fourth level. It should be noted that as shown in Fig. 8, minterm $x_1x_2x_3\overline{x_4}y_1$ is not covered by any dashed-line rectangles. This is because the minterm is not in the third level of the SICC-cube graph. Since we only require all dashed-line rectangles, i.e., cubes in set M_2, to cover all minterms in the third level, it is possible that a minterm not in the third level is not covered by any dashed-line rectangles. □

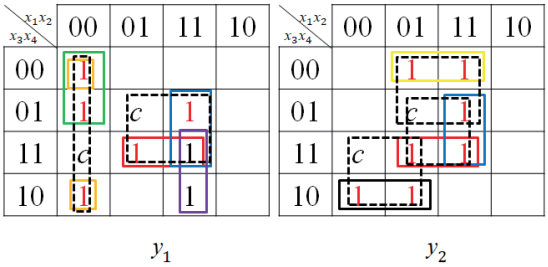


Figure 8: The same Karnaugh map as that shown in Fig. 1. The c's form the root node of the SICC-cube graph shown in Fig. 6. A dashed-line rectangle corresponds to a shaded cube in the second level of the graph. A red 1 corresponds to a minterm in the third level of the graph.

3) Calculating Literal Reduction: In this step, we calculate the literal reduction due to the addition of the cubes in set M_2 and the removal of the cubes in set M_1. It is shown in Line 14 in Alg. 4. The literal reduction is calculated by subtracting the total literal count of the cubes in set M_2 from the total literal count of the cubes in set M_1. For the example in Fig. 6, the literal reduction equals $26 - 13 = 13$.

It should be noted that as a byproduct of the proposed estimation method, it gives us a way to construct the approximate SOP expression that gives the estimated literal reduction.

TABLE I: The optimal subset of the prime SICCs for six benchmark circuits. Each cell gives the list of the sizes of the prime SICCs in the optimal subset.

NoE threshold	Benchmark					
	z9sym	sym10	rd73	clip	sao2	5xp1
3	1,2	1,1,1	1,1,1	1,2	1,1,1	2,1
4	1,2,1	1,1,1,1	1,1,1,1	1,1,2	1,1,1,1	1,2,1

Specifically, we can construct the approximate SOP expression by removing the cubes in set M_1 from the original expression and adding the cubes in set M_2 into the original expression.

V. Speed-up Techniques

A drawback of the heuristic search method is its long runtime, especially when the NoE threshold e is large. To estimate a lower bound on the runtime, we assume that the number of prime SICCs of size 1 is k. Then, the number of SICCs of size e formed by these size-1 prime SICCs is $\binom{k}{e}$. This number is a lower bound on the total number of subsets of the prime SICCs that are checked by our method. Assume that the average time of estimating the literal reduction of a subset of the prime SICCs is t. Thus, a lower bound on the runtime is $\binom{k}{e}t = \Omega(k^e t)$. Therefore, the runtime at least increases exponentially with the NoE threshold e. For the circuits that we tested, k is on the order of hundreds. Thus, the runtime is very large when the NoE threshold is large. In this section, we present four speed-up techniques to reduce the runtime.

A. Progressive Error Reduction

We propose a *progressive error reduction method (PERM)* based on the experiments of the original heuristic search method. We chose the NoE threshold as 3 and 4 and applied the original method to six benchmark circuits. The results are shown in Table I. Each cell shows the list of the sizes of the prime SICCs in the optimal subset. For example, when the NoE threshold is 3, the optimal subset for the benchmark `z9sym` consists of 2 prime SICCs. The first is of size 1 and the second is of size 2. From the table, we can see that although there exist prime SICCs of sizes 3 and 4, the optimal subset for each benchmark is only composed of prime SICCs of size 1 or 2. The reason for this can be explained as follows. Each prime SICC corresponds to a set of promising cubes that can be added into the original SOP expression. Thus, a set of prime SICCs of small sizes may produce more promising cubes than a single SICC of a large size. Consequently, the former can cover more existing cubes than the latter and hence, remove more literals. Therefore, the optimal subset tends to consist of prime SICCs of sizes 1 or 2.

Based on this observation, we propose PERM. The basic idea is to produce a sequence of approximate SOP expressions. A later expression is produced from a previous expression by further introducing an optimal SICC of either size 1 or size 2. This is done by calling the proposed heuristic search method on the previous expression with the NoE threshold as 2. As we discussed in Section IV-E, in the last step of the heuristic method, a result array R is obtained (see Line 1 of Alg. 3), in which $R[i].mlr$ records the maximal literal reduction among all prime SICC unions with size equal to i and $R[i].t$ records the combined SICC-cube tree that gives the maximal literal reduction. In PERM, we exploit the array R to generate two SOP expressions from the previous SOP expression F. The first is produced by introducing the SICC of $R[1].t$ into F, while the second is produced by introducing the SICC of $R[2].t$ into F. Each SOP expression is also associated with a remaining NoE. Compared to the previous SOP F, the remaining NoE of the first SOP is reduced by 1 and that of the second SOP is reduced by 2.

The entire flow of PERM is shown in Alg. 5. Assume that the NoE threshold is e. We maintain $(e+1)$ SOP sets $S_0, S_1, \ldots, S_e$, where S_i stores the SOPs with the remaining NoE as i. Initially, $S_0, \ldots, S_{e-1}$ are set as empty (see Lines 1–2) and S_e consists of the original SOP expression F (see Line 3). Then, we iterate from set S_e down to set S_1 (see Line 4). Line 5 at the beginning of each iteration is ignored for the current moment. For each SOP f in set S_i, we approximate it by Alg. 1 under the NoE threshold 2 and derive two SOPs f_1 and f_2 with the remaining NoEs as $(i-1)$ and $(i-2)$, respectively (see Line 7). Then, f_1 is added into set S_{i-1} (see Line 8) and f_2 is added into set S_{i-2} (see Line 9). After all SOPs in set S_i are processed, set S_{i-1} stores all SOPs with the remaining NoEs as $(i-1)$. Then, we continue with set S_{i-1}. Finally, after all the SOPs are built, we check all of them and pick the one that gives the fewest literals (see Line 10).

Algorithm 5: The progressive error reduction method with limited processed SOPs.

Input : a simplified SOP expression F, an NoE threshold e, and a parameter H.
Output : an approximate SOP expression F'.

1 **for** $i \leftarrow 0$ *to* $e-1$ **do**
2 | the set of SOPs with the remaining NoE as i, $S_i \leftarrow \varnothing$;
3 $S_e \leftarrow \{F\}$;
4 **for** $i \leftarrow e$ *down to* 1 **do**
5 | $S_i \leftarrow$ the top H SOPs with the fewest literals in S_i;
6 | **for** *each SOP* f *in* S_i **do**
7 | | approximate f by Alg. 1 under NoE threshold 2 and derive two SOPs f_1 and f_2 with the remaining NoEs as $i-1$ and $i-2$, respectively;
8 | | $S_{i-1} \leftarrow S_{i-1} \cup \{f_1\}$;
9 | | $S_{i-2} \leftarrow S_{i-2} \cup \{f_2\}$;
10 $F' \leftarrow$ the approximate SOP with the fewest literals in the set $S_e \cup S_{e-1} \cup \cdots \cup S_0$;
11 **return** F';

Fig. 9 shows an example where the NoE threshold is 4. It presents all SOPs built by PERM in the form of a tree. Each node in the tree corresponds to an SOP. Each edge connects two SOPs where the one at the end of the edge is derived from the one at the beginning of the edge. The value in each node is the remaining NoE for that SOP and the value near each edge is the reduced NoE, which is either 1 or 2. The final optimal approximate SOP is the one that gives the fewest literals in the tree.

Now, we analyze the time complexity of this method. From Alg. 5, it is obvious that the amount of work is dominated by the total amount of work of applying Alg. 1 to SOPs.

We first analyze the runtime of each invocation of Alg. 1. We assume that for each SOP processed by Alg. 1, its numbers of prime SICCs of sizes 1 and 2 are no more than k and l, respectively. The algorithm has three steps. In the first step, it builds the set of SICC-cube trees by traversing the Hasse diagram. Assume that the average time that the first step spends in visiting one cube in the Hasse diagram is τ. Given that the Hasse diagram has $3^m(2^n-1)$ cubes, the runtime of the first step is $O(3^m 2^n \tau)$. In the second step, the algorithm creates the set of augmented SICC-cube trees by checking all pairs of SICC-cube trees such that one has a prime SICC of size 1 and the other has a prime SICC of size 2. For each pair, if proper, it may merge the cubes in the second-level

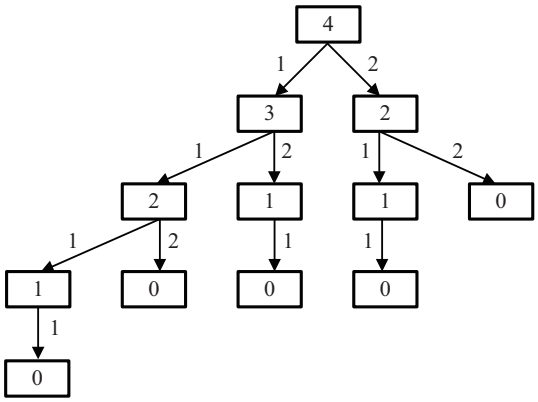


Figure 9: The solution tree built by the progressive error reduction method for an example with the NoE threshold of 4.

of the SICC-cube tree of a prime SICC of size 1 into the second-level of the SICC-cube tree of a prime SICC of size 2. Assume that the average time the second step spends for each pair is λ. Given that the number of pairs is no more than kl, the runtime of the second step is $O(kl\lambda)$. In the third step, the algorithm checks all subsets of the prime SICCs to produce two resulting SOPs. The number of unions of prime SICCs checked is no more than $k + \binom{k}{2} + l$, where the first k corresponds to the maximal number of prime SICC unions with sizes of 1 and the sum $\binom{k}{2}+l$ corresponds to the maximal number of prime SICC unions with sizes of 2. Assume that the average time of estimating the literal reduction of a union of prime SICCs is t. Then, the runtime of the third step is $O((k+\binom{k}{2}+l)t) = O((k^2+l)t)$. In summary, the runtime of each invocation of Alg. 1 is $O(3^m 2^n \tau + kl\lambda + (k^2+l)t)$. It should be noted that λ is much less than t. Thus, the runtime of each invocation of Alg. 1 can be simplified as

$$O(3^m 2^n \tau + (k^2+l)t). \tag{2}$$

The total number of invocations equals the number of SOPs processed. As Fig. 9 shows, if the NoE threshold is e, the number of SOPs processed equals the number of internal nodes in the tree, which is no more than the total number of nodes in a perfect binary tree[1] of e levels. Thus, the number of invocations equals $O(2^e)$ and the runtime of PERM is

$$O(2^e(3^m 2^n \tau + (k^2+l)t)).$$

As we analyzed before, the runtime of the heuristic search method is at least $\Omega(k^e t)$. Given that k is much larger than 2, when the NoE threshold e is larger than both the number of inputs m and the number of outputs n, PERM runs much faster than the heuristic search method.

B. Limiting the Processed SOPs

Although PERM runs much faster than the heuristic search method, its runtime still increases exponentially with the NoE threshold e. We propose an additional speed-up technique by limiting the processed SOPs. We call it *limiting processing (LP)* for short.

As the tree in Fig. 9 shows, the number of SOPs of the same remaining NoE grows with the remaining NoE. This causes the exponential increase of the runtime. To reduce the runtime, we introduce a time-quality tuning parameter H. Among all SOPs of the same remaining NoE, we only process the top H with the fewest literals to produce further SOPs. This is realized by Line 5 in Alg. 5: before we process each SOP in set S_i, which stores all SOPs with i remaining NoEs, we first shrink S_i by only keeping the top H SOPs in it with the fewest literals. Note that if the parameter H is set to infinity, it is the normal PERM.

With this technique, the total number of invocations of Alg. 1 is limited to eH. Thus, the runtime reduces to

$$O(eH(3^m 2^n \tau + (k^2+l)t)), \tag{3}$$

which only increases linearly with the NoE threshold. By setting different values for H, we can achieve a trade-off between the runtime and the design quality.

C. Reducing the Subsets of the Prime SICCs for Checking

PERM with LP reduces the number of invocations of Alg. 1. To further reduce the runtime, we can reduce the runtime of each invocation of Alg. 1. As we analyzed in Section V-A, the runtime of each invocation of Alg. 1 is dominated by two components, the runtime of the first step, which is $O(3^m 2^n \tau)$, and the runtime of the third step, which is $O((k^2+l)t)$. Note that t, the average time of estimating the literal reduction of a subset of the prime SICCs, is much larger than τ, the average time that the first step spends in visiting one cube in the Hasse diagram. When the circuit size is small (i.e., $3^m 2^n$ is small), the runtime of each invocation of Alg. 1 is dominated by that of the third step. In this section, we propose a technique to further reduce the runtime of the third step of Alg. 1. In PERM, each invocation of Alg. 1 has the NoE threshold as 2. Thus, one major work in the third step is checking the pairs of prime SICCs of sizes 1. Therefore, to decrease the runtime of the third step, we can decrease the number of pairs of prime SICCs of sizes 1 that we need to check.

To propose an acceleration strategy, we performed an empirical study of PERM on six benchmarks. We identified the sequence of SICCs that leads to the final solution. We found that when an SICC in the sequence is of size 2 and is formed by two prime SICCs of size 1, the first component SICC is ranked within top 21% in the entire list of size-1 prime SICCs by the literal reduction, while the second was ranked within top 72%. This is reasonable because the top-ranked size-1 prime SICCs reduce more literals than the others. Based on this observation, we propose the following strategy to reduce the number of size-1 prime SICC pairs for checking. We only consider the size-1 prime SICC pairs such that the first prime SICC is within top 25% and the second prime SICC is within top 80%. We call this speed-up technique *reducing subsets (RS)* for short.

Applying RS does not affect the runtime complexity equation of PERM with LP. Its runtime complexity equation is still Eq. (3). However, it reduces the number of size-1 primes SICC pairs that needs to be checked from $\frac{k(k-1)}{2}$ to roughly $0.2k^2$. Thus, this effectively reduces the runtime of the third step in Alg. 1. As will be demonstrated by the experimental results in Section VI, this speed-up technique does not reduce the final quality much.

D. Reducing the Number of Cubes Visited in the First Step of Alg. 1

The speed-up technique proposed in Section V-C reduces the runtime of the third step in Alg. 1. However, it is only effective when the circuit size is small. For a large circuit

[1]A perfect binary tree is is a binary tree in which all internal nodes have two children and all leaves have the same depth.

with large m and n, the exponential factor in the runtime complexity formula of the first step, i.e., 3^m2^n, is very large. For example, for a circuit with $m = 14$ and $n = 8$, the exponential factor 3^m2^n is more than 1.2 trillion. As a result, the runtime of the first step dominates that of Alg. 1. In this section, we propose a speed-up technique to handle a large circuit by reducing the runtime of the first step of Alg. 1. The key idea is to reduce the exponential factor.

The exponential factor 3^m2^n is due to the traversal of each cube in the Hasse diagram. Thus, to reduce this factor, we propose to reduce the number of cubes that we visit in the Hasse diagram. In the proposed method, we only visit the existing cubes in the original SOP and their ancestor cubes in the Hasse diagram to build the set of SICC-cube trees. This is achieved by starting from each existing cube and visiting its ancestors bottom up. To avoid unnecessary cube checking, when we visit an ancestor cube c and find that its number of EICs is larger than the given NoE threshold e, we will stop exploring the ancestor cubes of c. This is because any ancestor cube of c covers c and hence, its number of EICs must also be larger than the NoE threshold. By Definition 2, such a cube cannot belong to any SICC-cube tree. Note that this technique is used within PERM where the NoE threshold is set as 2 for each invocation of Alg. 1. Thus, we usually do not need to go very far from an existing cube to reach an ancestor with its number of EICs larger than the NoE threshold. Consequently, the number of cubes that we visit is significantly reduced, causing the runtime reduction for the first step of Alg. 1. Of course, this method may influence the quality of the results, since it may miss candidate cubes that are not ancestor cubes of any existing cube but can still reduce the literal count.

VI. Experimental Results

In this section, we present the experimental results on our proposed heuristic search method and the speed-up techniques. We implemented the algorithms in C++. In Sections VI-A–VI-F, the experiments were carried on six small benchmarks, `z9sym`, `sym10`, `rd73`, `clip`, `sao2`, and `5xp1`, which were also used in [24] and [26]. In Section VI-G, the experiments were carried on more benchmarks, including some larger ones. All the experiments were run on a desktop with a quad core I5-6500 3.2*GHz* CPU and 32*GB* RAM.

A. Proposed Literal Reduction Estimation Method

As mentioned in Section IV-F, one important procedure is to estimate the literal reduction of the union of a subset of the prime SICCs. In this section, we studied the effectiveness of our literal reduction estimation method proposed in Section IV-F. We compared it with the method based on Espresso mentioned at the beginning of that section. The approximation flow where these estimation methods are applied is the heuristic search method without any speed-up techniques. The results are shown in Table II. Column 1 of the table lists the benchmark names. The values after the letters "i", "o", and "c" are the number of inputs, the number of outputs, and the number of existing cubes, respectively, for a benchmark. Column 2 is the given NoE threshold. Columns 3, 4, and 6 list the numbers of literals for the original benchmark, the approximate version with our proposed literal reduction estimation method, and the approximate version with the Espresso-based method, respectively. Columns 5 and 7 list the runtime for estimating the literal reduction for all subsets of the prime SICCs by the proposed method and that by the Espresso-based method, respectively.

TABLE II: The comparison between our proposed literal reduction estimation method and the Espresso-based estimation method.

		Original	Proposed		Espresso	
Benchmark	NoE	Literals	Literals	Time/ms	Literals	Time/ms
z9sym	1	610	581	10	581	82
i:9,o:1,c:86	2		567	940	554	3734
sym10	1	1470	1344	116	1344	284
i:10,o:1,c:210	2		1292	23269	1302	20353
rd73	1	903	881	5	881	56
i:7,o:3,c:127	2		865	554	862	3872
clip	1	793	755	10	756	103
i:9,o:5,c:120	2		743	494	738	3962
sao2	1	496	459	35	458	403
i:10,o:4,c:58	2		429	3146	428	33020
5xp1	1	347	327	1	347	24
i:7,o:10,c:65	2		315	53	320	552

TABLE III: The comparison between the basic heuristic search method and PERM.

		Original	Heuristic		PERM	
Benchmark	NoE	Literals	Literals	Time/s	Literals	Time/s
z9sym	2	610	567	0.883	560	0.848
i:9,o:1	3		552	38.888	547	1.611
c:86	4		529	1444	530	3.042
sym10	2	1470	1292	23.625	1292	24.305
i:10,o:1	3		1269	1985	1259	48.428
c:210	4		1245	122674	1227	93.72
rd73	2	903	865	0.586	865	0.600
i:7,o:3	3		849	39.984	849	1.167
c:127	4		833	2917	833	2.248
clip	2	793	743	1.458	738	2.395
i:9,o:5	3		729	11.212	721	4.878
c:120	4		717	333	706	8.622
sao2	2	496	429	2.399	429	3.801
i:10,o:4	3		408	62.743	408	7.571
c:58	4		395	3725	394	13.068
5xp1	2	347	315	4.345	311	6.859
i:7,o:10	3		312	5.105	302	14.114
c:65	4		300	23.683	289	25.003

According to Table II, the quality of the approximate expression using our literal reduction estimation method is similar to that using the Espresso-based method. However, our method is $9\times$ faster than the Espresso-based method on average. This shows the effectiveness of our proposed literal reduction estimation method. In the following experiments, we will use this method for estimating the literal reduction.

B. Progressive Error Reduction Method (PERM)

In this section, we studied the effectiveness of PERM proposed in Section V-A. We compared it with the basic heuristic search method. The results are shown in Table III. This table is similar to Table II, except that the time refers to the entire runtime of a method. In terms of the quality, PERM can decrease more literals than the heuristic search method for most cases. This is because the former works on a sequence of approximation problems of smaller NoE thresholds, while the latter works on a single problem of a larger NoE threshold. Thus, the former may give more accurate literal reduction estimation than the latter, and hence, a better quality. In terms of the runtime, although PERM spends a little more time than the heuristic search method for NoE thresholds smaller than 3, for a larger NoE threshold, it saves much more time for most cases. For NoE threshold of 4, it can reduce 81.9% runtime on average. The trend of runtime increase over the NoE threshold agrees with our runtime analysis in Section V-A: the runtime of PERM roughly increases in the same trend as 2^e, where e is the NoE threshold. Although it is still exponential, the speed of runtime increase is much slower than that of the heuristic search method. In summary, PERM is superior to the heuristic search method for a large NoE threshold.

TABLE IV: The comparison among PERM with and without LP and with and without RS.

Bench.	NoE	PERM		$H=2$		RS		RS, $H=2$	
		Literals	Time/s	Literals	Time/s	Literals	Time/s	Literals	Time/s
z9sym i:9,o:1	2	560	0.848	560	0.856	560	0.263	560	0.283
	4	530	3.04	525	3.66	530	0.913	525	0.841
	8	450	22.3	454	9.54	450	6.58	455	2.01
	16	323	862	324	19.4	320	284	335	4.20
sym10 i:10,o:1	2	1292	24.3	1292	23.9	1294	1.10	1294	1.14
	4	1227	93.7	1227	106	1228	3.86	1230	3.77
	8	1098	711	1099	317	1100	28.3	1100	9.13
	16	845	26071	845	673	847	1169	851	18.4
rd73 i:7,o:3	2	865	0.600	865	0.602	865	0.227	865	0.235
	4	833	2.25	833	2.52	820	0.782	820	0.804
	8	769	17.3	769	7.11	727	5.63	728	2.00
	16	589	624	596	15.0	571	214	573	3.83
clip i:9,o:5	2	738	2.40	738	2.40	738	2.22	738	2.35
	4	706	8.62	706	8.98	703	7.82	703	7.92
	8	653	64.7	653	21.9	652	60.7	652	19.2
	16	584	2906	584	45.9	579	2886	579	40.5
sao2 i:10,o:4	2	429	3.80	429	3.79	429	3.05	429	3.20
	4	394	13.1	394	13.7	394	10.8	394	10.7
	8	322	86.3	326	32.4	318	82.6	318	25.5
	16	209	4079	221	65.7	209	3924	209	53.1
5xp1 i:7,o:10	2	311	6.86	311	6.84	311	6.82	311	7.04
	4	289	25.0	294	26.8	294	24.7	294	26.0
	8	264	199	266	67.7	264	195	266	66.2
	16	227	9222	228	149	225	6935	228	143

C. Limiting the Processed SOPs (LP)

In this section, we studied the effectiveness of the second speed-up technique, LP, which is proposed in Section V-B. We compared PERM with and without LP. The results of PERM with and without LP are shown in columns 5 and 6 under the title "$H=2$" and columns 3 and 4 under the title "PERM" in Table IV, respectively. When LP is applied, we set the time-quality tuning parameter H to 2. From the table, we can see that applying LP accelerates PERM for NoE threshold larger than or equal to 8. The speed-up ratio grows with the NoE threshold. For NoE threshold of 16, the speed-up ratio reaches up to $63\times$. For benchmark `clip`, setting the parameter H to 2 does not affect the approximation quality for all the tested NoE thresholds. For the remaining benchmarks, applying LP degrades the quality slightly for a large NoE threshold. For NoE threshold of 16, the average quality reduction over the remaining benchmarks is only 1.5%. Overall, LP is quite effective.

D. Reducing Subsets of the Prime SICCs for Checking (RS)

In this section, we studied the effectiveness of the third speed-up technique, RS, which is proposed in Section V-C. The experimental results are also shown in Table IV. In the table, columns 7 and 8 under the title "RS" show the quality and runtime, respectively, of PERM with RS. Comparing column 4 with column 8 we can see that PERM with RS runs faster than PERM without RS for all the tested NoE thresholds. On average, applying RS can reduce 42.4% runtime. Comparing column 3 with column 7 we can see that applying RS does not degrade the quality in most cases. For many cases, applying RS can even improve the quality. This can be explained as follows. With RS, the subsets formed by the prime SICCs in the later part of the ordered SICC lists are not considered. However, if they are considered, they have the potential of giving a better literal reduction estimation. However, due to the inexactness of the literal reduction estimation, they are actually not the good choices. Thus, RS helps eliminate these "false" optimal subsets.

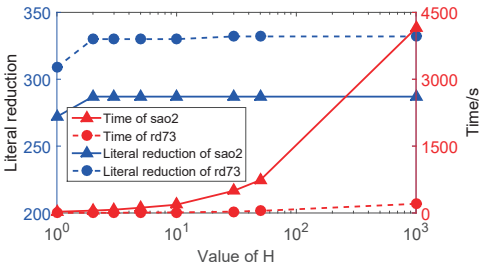


Figure 10: The effect of the parameter H on the runtime and the quality for PERM with LP and RS.

E. PERM with both LP and RS

In this section, we studied the effectiveness of PERM with both LP and RS. LP has a time-quality tuning parameter H. To decide a good choice for H, we first tested the method on two benchmarks `rd73` and `sao2` for different choices of H as $1, 2, 3, 5, 10, 30, 50$, and $+\infty$. The NoE threshold was chosen as 16. The results are shown in Fig. 10, where the blue curves show the literal reduction trend with the y-axis on the left, while the red curves show the runtime trend with the y-axis on the right. The dashed lines and the solid lines give the results for `rd73` and `sao2`, respectively. From Fig. 10, it is clear that the runtime of the method increases with H. The quality of approximate results improves a lot when H changes from 1 to 2. However, when H further increases, the quality improvement is very small. Based on this result, a good choice for H is 2.

Columns 9 and 10 under the title "RS, $H=2$" in Table IV list the quality and runtime, respectively, of PERM with both LP and RS and with parameter $H=2$. Comparing the columns under the title "PERM" and the columns under the title "RS, $H=2$" in Table IV, we can see that applying both LP and RS to PERM can significantly reduce the runtime. For the NoE thresholds of 8 and 16, the average speed-up ratios are $18\times$ and $333\times$, respectively. Meanwhile, the quality loss is small. Out of the total 24 test cases, only 9 has quality loss. The maximal quality loss is 3.7%. For some cases, the quality even improves. For example, for the benchmark `rd73` and the NoE threshold 8, the quality improves by 5.3%.

Finally, we studied how the runtime increases with the NoE threshold for PERM with both LP and RS. The parameter H was set as 2. Fig. 11 shows the runtime-versus-NoE-threshold curves for NoE thresholds of $1, 2, 4, 8, 16$ for six benchmarks. We can see that the runtime increases linearly with the NoE threshold for all benchmarks when the NoE is larger than 1. This agrees with our time complexity analysis in Section V-C: as shown in Eq. (3), the runtime of PERM with LP and RS increases linearly with the NoE threshold e.

In summary, the experimental results in this section demonstrate the advantage of applying both LP and RS to PERM. Thus, we will use PERM with LP and RS to compare with other state-of-the-art methods.

F. Comparison with Other Methods

In this section, we compared our proposed PERM with LP and RS to three other methods: the exhaustive search for the optimal result, the method proposed in [24], and the method proposed in [26]. We set the time-quality tuning parameter H to 2. The method proposed in [24] is a state-of-the-art method handling the same problem as ours. The comparison results are shown in Table V. Since the exhaustive search method has a very high time complexity, we only did the comparison

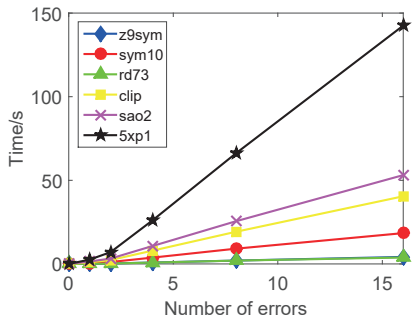


Figure 11: Runtime versus NoE threshold for PERM with LP and RS.

TABLE V: Comparison among our method, the exhaustive search method, the method in [24], and the method in [26].

Bench.	NoE	Literal count					Time/s		
		Original*	Optimal	[24]	[26]	Our*	Optimal	[26]	Our
z9sym	1	610	581	581	581	**581**(84)	0.883	0.014	0.103
i:9,o:1	2	(86)	554	564	572	**560**(81)	41.7	0.016	0.283
sym10	1	1470	1344	1345	1345	**1344**(210)	2.94	0.042	0.45
i:10,o:1	2	(210)	1290	1290	1312	1294(210)	244	0.047	1.14
rd73	1	903	873	886	887	**881**(124)	1.45	0.004	0.034
i:7,o:3	2	(127)	843	866	875	**865**(121)	191	0.015	0.235
clip	1	793	755	777	777	**755**(114)	62.8	0.026	0.965
i:9,o:5	2	(120)	738	759	775	**738**(112)	86525	0.039	2.35
sao2	1	496	447	482	464	**459**(53)	17.3	0.061	1.46
i:10,o:4	2	(58)	408	-	455	**429**(49)	21328	0.084	3.20
5xp1	1	347	324	339	347	**327**(62)	66.9	0.006	2.59
i:7,o:10	2	(65)	305	314	347	**311**(61)	300323	0.012	7.04

* A value inside a pair of parentheses is the number of cubes in the corresponding simplified SOP.

under NoE thresholds of 1 and 2. Column 4 in the table shows the optimal results returned by the exhaustive search. Columns 5 and 6 are the experiment results from [24] and [26], respectively. Column 7 shows the results of our method. For each test case, if our result is no worse than the corresponding results in [24] and [26], we highlight it in bold.

From the table, we can see for single-output functions, i.e., `z9sym` and `sym10`, our method and the method in [24] have nearly the same quality and both are near the optimal. For multiple-output functions, our method is better than the methods in [24] and [26]. This is because the methods in [24] and [26] partition a multiple-output function into multiple single output functions. Thus, they lose the optimization opportunity across multiple outputs. In contrast, our method considers the multiple outputs synergistically. When only the multiple-output functions are considered, the average literal reduction ratios of our method over the methods in [24] and [26] are 2.22% and 4.04%, respectively, while the maximal literal reduction ratios of our method over those in [24] and [26] are 4.77% and 10.3%, respectively. It should be noted that in this set of experiments, the NoE threshold is very small (i.e., 1 or 2). Thus, in some cases, the other methods already produce results very close to the optimal results shown in column 4 of the table. Thus, the additional space for improvement is limited and it causes the reduction ratio of our method to be small. However, on the other hand, further closing the *small* gap between the result of an existing method and the optimal result is a challenging task, but our method is able to achieve it. Thus, it still shows the value of the proposed method. Another important point showing the effect of our method is that for benchmark `clip`, it achieves the exact optimal result, while the other methods cannot.

The last three columns list the runtime of the exhaustive search, the method in [26], and our method, respectively. Since we do not have the source code of the method in [24], we do not list the runtime of it. Compared with the exhaustive search, our method is much faster, which is expected. Compared with the method in [26], our method is slower. This is because the method in [26] does not estimate the literal reduction and it simplifies the procedure by partitioning a multiple-output function into individual single-output functions. However, the runtime of our method is still within ten seconds and it has better quality than the method in [26] for all the test cases.

Finally, we compared our method with the methods in [24] and [26] for large NoE thresholds. In [24], the authors plotted a literal-reduction-ratio-versus-error-rate curve for NoE thresholds of $1, 2, 4, 8$. We did the same thing here. The comparison for single-output benchmarks is shown in Fig. 12a and that for multiple-output benchmarks is shown in Fig. 12b. In both figures, we use solid lines to indicate the results of [24], dashed lines to indicate the results of [26], and dash-dotted lines to indicate our results. The results of the three methods for the same benchmark are shown in the same color. As shown in Fig. 12a, for single-output functions, our method reduces a similar number of literals as the methods in [24] and [26] for NoE thresholds no more than 4. However, for a larger NoE threshold like 8, our method is better. As shown in Fig. 12b, for multiple-output functions, our method reduces more literals than the methods in [24] and [26] for all the ERs. Generally, the improvement by our method increases as the ER increases. Notably, for benchmark `sao2`, when the ER is 0.8%, our method can reduce 36% literals over the input expression optimized by Espresso, while the methods in [24] and [26] can only reduce 23% and 13% literals, respectively. On average, for an ER threshold of 0.8%, the methods in [24] and [26] can only reduce 11.4% and 8.97% literals, respectively, while our method reduces 15.8% literals.

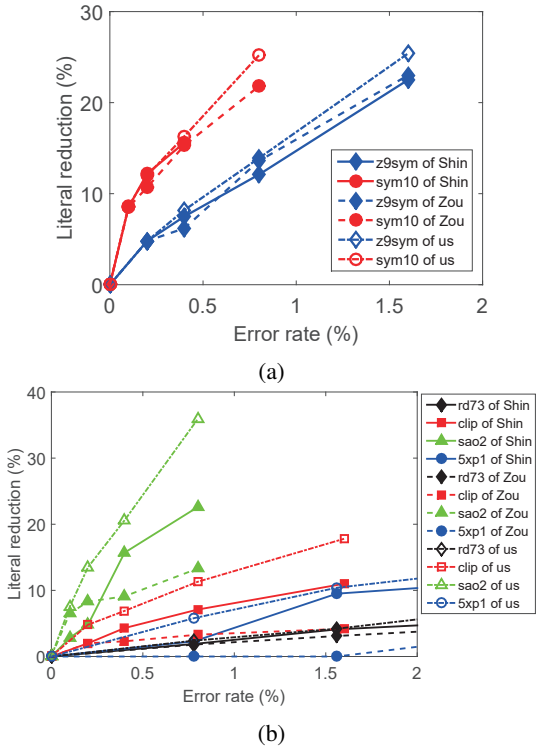


Figure 12: Comparison between our method and the methods in [24] (called Shin's method) and [26] (called Zou's method) on literal reduction for different error rates: (a) single-output functions; (b) multiple-output functions.

G. Results on More Benchmarks

TABLE VI: The basic information of 19 benchmarks in the IWLS93 benchmark suit and the approximate results by our method under the NoE threshold of 16.

Benchmark	Original		Approximate			Literal
	Literals	Cubes	Literals	Cubes	Time/s	Reduction(%)
con1, i:7, o:2	32	9	32	9	0.385	0
rd73, i:7, o:3	903	127	578	88	1.48	36.0
inc, i:7, o:9	198	30	156	25	0.494	21.2
5xp1, i:7, o:10	347	65	235	49	0.726	32.3
sqrt8, i:8, o:4	188	38	98	22	0.587	47.9
rd84, i:8, o:4	2070	255	1578	218	6.52	23.8
misex1, i:8, o:7	96	12	96	12	0.509	0
z9sym, i:9, o:1	610	86	340	54	2.59	44.3
chip, i:9, o:5	793	120	588	93	1.99	25.9
apex4, i:9, o:19	5419	436	5040	421	109	7.0
sao2, i:10, o:4	496	58	231	29	2.48	53.4
ex1010, i:10, o:10	2718	284	2693	283	14.3	0.920
alu4, i:14, o:8	5087	575	4904	562	298	3.60
misex3, i:14, o:14	7784	690	7446	656	693	4.34
table3, i:14, o:14	2644	175	2459	165	513	7.0
misex3c, i:14, o:14	1561	197	1239	163	252	20.6
b12, i:15, o:9	207	43	207	43	249	0
t481, i:16, o:1	5233	481	5105	473	1570	2.45
table5, i:17, o:15	2501	158	2410	154	7868	3.64
Average	2046	202	1865	185	610	17.6

In the previous works [24] and [26], only a limited number of benchmarks were tested. To benefit future research on the same topic, in this section, we present the experimental results of our proposed method on more benchmarks. We applied our method to the circuits in the IWLS93 benchmark suit [33]. Due to the memory limitation, the algorithm can only handle circuits with fewer than 20 inputs and the sum of the numbers of inputs and outputs fewer than 34. Furthermore, we did not consider circuits with fewer than 6 inputs, since for these circuits, a small NoE will give a large ER. As a result, we tested our method on 19 circuits as shown in Table VI. Column 1 of the table lists the circuit names, the numbers of inputs (i), and the numbers of outputs (o). For those circuits with more than 10 inputs, even PERM with both LP and RS is slow due to the prohibitively large Hasse diagram. Thus, our method, which was applied to all the circuits, further includes the speed-up technique proposed in Section V-D. The time-quality tuning parameter H was set as 2 here. Columns 2 and 3 of the table list the numbers of literals and cubes of the original SOPs, respectively. Columns 4 and 5 list the numbers of literals and cubes of the approximate results by our method, respectively, under the NoE threshold of 16. Column 6 shows the runtime of our method. Column 7 lists the literal reduction rates.

Among 19 circuits, our method cannot reduce any literals for three circuits, i.e., `con1`, `misex1`, and `b12`. This is because these circuits do not have any prime SICCs of sizes 1 or 2 and the speed-up technique PERM reduces literals only when a circuit has such prime SICCs. For some remaining circuits, a significant amount of literal reduction can be achieved. For example, for circuit `sao2`, the NoE of 16 corresponds to an ER of 1.56%. Under this ER, our method could reduce 53.4% literals. For circuit `misex3c`, the NoE of 16 corresponds to an ER of 0.10%. Under this ER, our method could reduce 20.6% literals. On average, under the NoE threshold of 16, our method reduces literals by 17.6%.

For circuits with at most 10 inputs, our method is fast: the runtime is no more than $15s$. However, as expected, the runtime of our method increases with the circuit size. For circuits with at least 14 inputs, our method takes more than $200s$. However, except the largest circuit `table5`, our method could still finish within half an hour. The average runtime over all 19 circuits is $610s$. If `table5` is excluded, the average runtime is $206s$.

By comparing Tables IV and VI, we can also see the effect of the speed-up technique proposed in Section V-D, i.e., reducing the number of cubes visited in the first step of Alg. 1. Specifically, Tables IV and VI share five common benchmarks, `z9sym`, `rd73`, `clip`, `sao2`, and `5xp1`. The last two columns of Table IV give the literal counts and runtimes of applying the method with all the speed-up techniques except the one proposed in Section V-D to these five circuits under the NoE threshold of 16. By comparing the corresponding entries in Tables IV and VI on these five circuits, we found that on average, using the speed-up technique proposed in Section V-D, we can further reduce the time by 77.9% with a small quality loss of 3.50%.

Fig. 13 further plots the average literal reduction over all circuits with the same number of inputs for different ERs. In this experiment, we excluded circuits `con1`, `misex1`, and `b12`, for which our method is ineffective. We can see that the average literal reduction increases with the ER. For the circuits with 9 inputs, our method reduces 40.8% literals on average under an ER threshold of 6.25%. For the circuits with 10 inputs, our method reduces 33.1% literals on average under an ER threshold of 3.12%. For the circuits with 14 inputs, our method reduces 13.1% literals on average under an ER threshold of 0.195%.

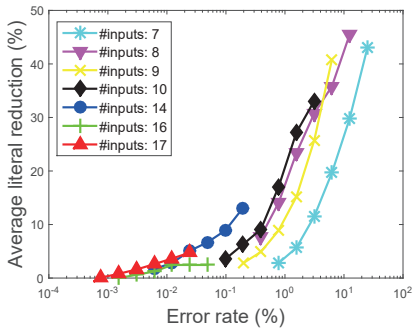


Figure 13: The average literal reduction over all circuits with the same number of inputs for different error rates.

VII. Conclusion and Future Work

In this paper, we presented a novel heuristic search method for two-level approximate logic synthesis under the error rate (ER) constraint. The key idea of our method is to search for an optimal set of input combinations for 0-to-1 output complement. We also proposed four speed-up techniques to reduce the runtime significantly. The experimental results showed that our accelerated search method is more effective than the previous state-of-the-art methods, especially for multiple-output circuits. In our future work, we will extend our method to handle the ER and the worst case error constraints together and handle incompletely specified Boolean functions with don't cares. We will also study how to apply the proposed method as a subroutine for multi-level approximate logic synthesis. Another direction to pursue is approximate logic synthesis for look-up table (LUT)-based FPGAs. In this case, since a LUT can implement any Boolean function of k inputs, minimizing the number of literals in the SOP does not readily translate into improvement in the LUT count. Thus, a different technique should be developed.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61574089 and the Natural Sciences and Engineering Research Council of Canada (NSERC) under Project No. RES0025211.

REFERENCES

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *European Test Symposium*, 2013, pp. 1–6.
[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.
[3] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.
[4] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, 2004.
[5] N. Zhu, W. L. Goh *et al.*, "Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1225–1229, 2010.
[6] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems," in *International Conference on Computer-Aided Design*, 2013, pp. 130–137.
[7] J. Hu and W. Qian, "A new approximate adder with low relative error and correct sign calculation," in *Design, Automation and Test in Europe*, 2015, pp. 1449–1454.
[8] C.-H. Lin and I.-C. Lin, "High accuracy approximate multiplier with error correction," in *International Conference on Computer Design*, 2013, pp. 33–38.
[9] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Design, Automation and Test in Europe*, 2014, pp. 95:1–95:4.
[10] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *International Conference on VLSI Design*, 2011, pp. 346–351.
[11] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *Design, Automation and Test in Europe*, 2011, pp. 1–6.
[12] S. Venkataramani, A. Sabne *et al.*, "SALSA: Systematic logic synthesis of approximate circuits," in *Design Automation Conference*, 2012, pp. 796–801.
[13] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Auntomation and Test in Europe*, 2013, pp. 1367–1372.
[14] A. Bernasconi and V. Ciriani, "2-SPP approximate synthesis for error tolerant applications," in *Euromicro Conference on Digital System Design*, 2014, pp. 411–418.
[15] J. Miao, A. Gerstlauer, and M. Orshansky, "Multi-level approximate logic synthesis under general error constraints," in *International Conference on Computer-Aided Design*, 2014, pp. 504–510.
[16] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Design Automation Conference*, 2016, pp. 128:1–128:6.
[17] A. Chandrasekharan, M. Soeken *et al.*, "Approximation-aware rewriting of AIGs for error tolerant applications," in *International Conference on Computer-Aided Design*, 2016, pp. 83:1–83:8.
[18] Y. Wu, C. Shen *et al.*, "Approximate logic synthesis for FPGA by wire removal and local function change," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 163–169.
[19] M. Traiola, A. Virazel *et al.*, "Towards digital circuit approximation by exploiting fault simulation," in *IEEE East-West Design and Test Symposium*, 2017, pp. 1–7.
[20] Y.-A. Lai, C.-C. Lin *et al.*, "Efficient synthesis of approximate threshold logic circuits with an error rate guarantee," in *Design, Automation and Test in Europe*, 2018, pp. 409–414.
[21] S. Su, Y. Wu, and W. Qian, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *Design Automation Conference*, 2018, pp. 54:1–54:6.
[22] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate logic synthesis using Boolean matrix factorization," in *Design Automation Conference*, 2018, pp. 55:1–55:6.
[23] A. Ranjan, A. Raha *et al.*, "ASLAN: Synthesis of approximate sequential circuits," in *Design, Automation and Test in Europe*, 2014, pp. 364:1–364:6.
[24] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Design, Automation and Test in Europe*, 2010, pp. 957–960.
[25] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *International Conference on Computer-Aided Design*, 2013, pp. 779–786.
[26] C. Zou, W. Qian, and J. Han, "DPALS: A dynamic programming-based algorithm for two-level approximate logic synthesis," in *International Conference on ASIC*, 2015, pp. 1–4.
[27] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.
[28] V. Mrazek, R. Hrbacek *et al.*, "EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation and Test in Europe*, 2017, pp. 258–261.
[29] R. K. Brayton and F. Somenzi, "An exact minimizer for Boolean relations," in *International Conference on Computer-Aided Design*, 1989, pp. 316–319.
[30] D. Baneres, J. Cortadella, and M. Kishinevsky, "A recursive paradigm to solve Boolean relations," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 512–527, 2009.
[31] B. Garrett, *Lattice Theory Revised*. American Mathematical Society, 1948.
[32] R. K. Brayton, G. D. Hachtel *et al.*, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
[33] K. McElvain, "IWLS93 benchmark set: Version 4.0," 1993. [Online]. Available: https://ddd.fit.cvut.cz/prj/Benchmarks/IWLS93.pdf

Sanbao Su is a master student in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. He received his B.S. degree from School of Management and Engineering at Nanjing University. His main research interests include logic synthesis for approximate computing.

Chen Zou is a Ph.D. student in the Computer Science Department of the University of Chicago. Chen Zou obtained B.S. in Microelectronics in June 2016 at Fudan University where he did some work on electronic design automation. Chen Zou lays a broad interest in different layers of the computer systems.

Weijiang Kong received the B.S. degree from Shanghai Jiao Tong University, Shanghai, China, in 2018. He is currently pursuing the masters degree at Department of Electronics, Royal Institute of Technology, Stockholm, Sweden. His current research interests include VLSI design and reconfigurable hardware architecture.

Jie Han (S'02-M'05-SM'16) is currently an Associate Professor in the Department of Electrical and Computer Engineering at the University of Alberta, Canada. He received the B.S. degree in Electronic Engineering from Tsinghua University in 1999 and the Ph.D. degree from the Delft University of Technology in 2004. His research interests include approximate computing, stochastic computing, reliability and fault tolerance, nanoelectronic circuits and systems, novel computational models for nanoscale and biological applications.

Weikang Qian (S'08-M'11) is an associate professor in the University of Michigan-Shanghai Jiao Tong University Joint Institute at Shanghai Jiao Tong University. He received his Ph.D. degree in Electrical Engineering at the University of Minnesota in 2011 and his B.Eng. degree in Automation at Tsinghua University in 2006. His main research interests include electronic design automation and digital design.