A Dynamic Expansion Order Algorithm for the SAT-based Minimization

Chia-Chun Lin, Kit Seng Tam, Chang-Cheng Ko, Hsin-Ping Yen, Sheng-Hsiu Wei,
Yung-Chih Chen, and Chun-Yao Wang, *Member, IEEE*

***Abstract*—Logic minimization attracted much attention in the early days because it is the engine for logic synthesis and optimization. Recently, a previous work proposed a SAT-based minimization algorithm for the patch function in the Engineering Change Order (ECO) problem. However, the algorithm is time-consuming for the functions in high dimension Boolean space. Therefore, in this paper, we propose an efficient algorithm that is suitable for the functions in high dimension Boolean space.**

***Index Terms*—Digital circuit, Logic optimization, SAT problem, High dimension Boolean space.**

I. Introduction

Logic optimization plays an important role in electronic design automation (EDA). This is because it simplifies logic circuits and leads a better Quality of Result (QoR) in terms of area, delay, etc. Although logic optimization is an NP-hard problem [11] [18], researchers have gained tremendous success in both two-level [4] [9] [10] [12] and multi-level [2] [3] [5] [14] [15] logic optimizations in practice.

Espresso [10] is a classic algorithm for two-level logic minimization. As compared with Quine-McCluskey minimization algorithm [9], Espresso adopts several heuristics to avoid the explosion issue of prime implicant as the number of inputs increases. Espresso consists of the following steps. First, *expand* each minterm to be a prime implicant. Second, *remove* the redundant cubes that have been covered by other prime implicants to get a prime cover. Third, *reduce* each prime implicant to a smaller cube. Then, in the next iteration, expand the cubes again in different directions. By applying these operations iteratively, an optimized two-level netlist can be obtained.

In addition to Espresso, the work of [13] also proposed a recursive SAT-based two-level minimization algorithm that minimizes the number of cubes in the sum-of-product (SOP) for computing patches in Engineering Change Order (ECO) recently. However, when the input dimension of target function grows to dozens to hundreds, the efficiencies of Espresso and recursive SAT-based algorithm [13] decrease. In fact, the efficiencies of these optimization algorithms can be improved significantly if we skip the *reduce* operation. That is, expanding each cube in a certain order until this cube becomes a prime implicant. The minimization procedure is terminated when all the minterms in the on-set have been covered by the prime implicants.

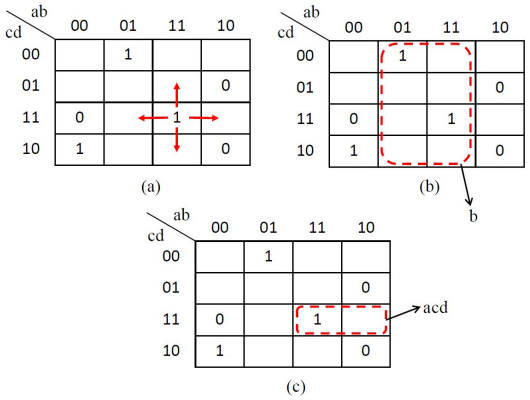


Fig. 1. (a) A function to be minimized. (b) A prime obtained by expanding the minterm 1111 with the expansion order $a > d > c > b$. (c) A prime obtained by expanding the minterm 1111 when the expansion order starts from b.

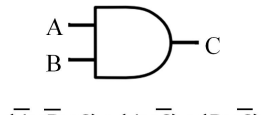
In this minimization procedure without the *reduce* operation, the expansion order critically influences the QoR. For example, given a Karnaugh map of an incompletely specified function as shown in Fig. 1(a). This function has three minterms, 0100, 0010, and 1111, in the on-set, and three minterms, 1001, 0011, and 1010, in the off-set. The other minterms are in the don't care-set. To obtain a minimized SOP form of the function, the minimization procedure expands a minterm in the on-set to a prime implicant iteratively. For an n-input function, unfortunately, there are $n!$ different orders to expand a minterm in the on-set. For example, the prime implicant b in Fig. 1(b) can be obtained by expanding the minterm 1111 with the expansion order $a > d > c > b$. However, this minimization procedure cannot reach a maximal prime implicant from the minterm 1111 when the expansion order starts from b, as shown in Fig. 1(c). This is because minterns in the off-set block the opportunities of impliciant acd for further optimization. Therefore, a good expansion order in this procedure becomes quite important and influences the size of resultant netlist significantly.

The problem formulation of this work is as follows: Given a simulation model of a design, we would like synthesize a minimized netlist that matches the behavior of the design. If the number of inputs is not large, i.e., smaller than or equal to 25, we can enumerate its truth table, and the synthesized netlist is exact from the viewpoint of functionality. However, if the number of inputs is larger than 25, or we say the function is in high dimension space, the truth table cannot be completely enumerated in practice. In this situation, we randomly exercise the simulation model to obtain the on-set and off-set patterns of

This work is supported in part by the Ministry of Science and Technology of Taiwan under MOST 106-2221-E-007-111-MY3, MOST 108-2218-E-007-061, MOST 109-2221-E-007-082-MY2, and MOST 109-2221-E-155-047-MY2.

C.-C. Lin, K. S. Tam, C.-C. Ko, H.-P. Yen, S.-H. Wei, and C.-Y. Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 30013, R.O.C. (email: chiachunlin@gapp.nthu.edu.tw; kitseng19960614@gmail.com; jamesko1026@gmail.com; amypin.yen@gmail.com; shenxiu5651@gmail.com; wcyao@cs.nthu.edu.tw). Y.-C. Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan 32003, R.O.C. (email: ycchen.cse@saturn.yzu.edu.tw).

2020 IEEE 33rd International System-on-Chip Conference (SOCC) | 978-1-7281-8746-4/20/$31.00 ©2020 IEEE | DOI: 10.1109/SOCC49529.2020.9524758



$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$

(a)

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

(b)

Fig. 2. (a) An AND gate and its corresponding CNF formula. (b) The truth table of AND function.

the design. The remaining patterns that are not simulated will be considered as don't cares in the procedure. As a result, the functionality of the synthesized circuit may be approximated to the original design, where the accuracy of the synthesized circuit is associated with the number of simulated patterns.

II. Preliminaries

A. Boolean satisfiability

Boolean satisfiability (SAT) problem is the problem of determining if there exists an assignment that satisfies the given Conjunctive Normal Form (CNF) formula. In 1990s, Marques-Silva et al. [1] [7] [8] proposed a conflict-driven clause learning algorithm, which can improve the performance of modern SAT solvers [22] [23] [24]. Due to the rapid progress on the performance of SAT solvers, many EDA problems can be solved effectively and efficiently when they are modeled as a SAT problem.

B. Tseitin transformation

Tseitin transformation algorithm [17] takes a combinational circuit as input and generates a corresponding CNF formula. Fig. 2(a) shows an AND gate and its corresponding CNF formula. Fig. 2(b) lists the truth table of AND function. We can see that only four assignments 000, 010, 100, and 111 satisfy the CNF formula. In other words, the CNF formula is identical to the function of AND gate. In fact, all the logic gates can be expressed as their corresponding CNF formulae.

C. SAT-based two-level minimization

In the traditional ECO patch generation flow, the first step is to find the on-set and the off-set of the patch function. Next, the on-set and off-set will be transformed into the corresponding CNF formulae, which are denoted as C_{on} and C_{off}, respectively, by the Tseitin transformation algorithm. Then, the resultant netlist of patch function can be synthesized by a SAT-based two-level minimization procedure.

Similar to minimizing a given function represented by Karnaugh map, the SAT-based minimization procedure iteratively expands the minterms in the on-set into prime implicants as follows. First, deriving a satisfying assignment from C_{on} using SAT solvers. Obviously, this satisfying assignment represents a minterm in the on-set of the patch function. The objective is to expand the minterm into a larger prime implicant by assigning some of the input bits as don't care. However, the obtained larger implicant cannot overlap the minterms in the off-set. Hence, we need to test if C_{off} has a satisfying assignment that is covered by the obtained implicant. When the result of test is UNSAT, which means that the off-set of patch does not overlap this larger implicant, this expansion is valid; otherwise, invalid. Once a valid prime implicant is found successfully, it will be removed from C_{on} such that the succeeding iterations will not repeatedly obtain an assignment that has been covered by this prime implicant. When we cannot obtain a satisfying assignment from C_{on} anymore, an irredundant prime SOP is derived.

The authors in [13] proposed a recursive algorithm to expand the selected minterms for minimizing the number of prime implicants in the synthesized netlist. The inputs of the recursive algorithm is the CNF formula of off-set, C_{off}, and a set of input variables obtained by the satisfying assignment from C_{on}. The algorithm will return the input variables that are needed in the implicant. In other words, the input variables not returned are don't care variables after the algorithm. However, the recursive algorithm is very time-consuming for functions in high dimension space. Therefore, we propose an improved approach focusing on the logic optimization in high dimension space in this work. More details about this approach will be presented in Section III.C.

III. Logic Optimization Framework

A. Handling multiple-output functions

The dimension of input space is a critical issue in this problem. Once the number of inputs in the given function is smaller than or equal to 25, this problem becomes easier from the viewpoint of analysis accuracy. That is, the synthesized circuit will not suffer from accuracy loss as we can completely enumerate the truth table. Since the input of this work is a simulation model, we do not have the netlist of function such that we cannot structurally identify the transitive fanin cone of each output for reducing the input dimension in a given multiple-output function. Thus, we decompose a function into many single-output functions by observing one output bit in the simulated results at a time. Next, we use another method, which will be explained in Section III.B, to identify the relevant inputs with respect to an output.

B. Functional pruning

After simulating a set of random patterns for the single-output function obtained from Section III.A, we may find that a set of inputs I are not relevant to the output O. That is, the value of O in this simulation model is intact when each input $\in I$ toggles under all simulated random patterns. Thus, we assume that this set of inputs I are not within the fanin cone of output O, which are called *dummy inputs* in this work, and we temporarily remove them in the succeeding minimization procedure. Before explaining how to identify the dummy inputs, we define the Relevance Index (RI) of an input first.

Definition 1: Given an input pattern p of a single-output function f, we define $flip(p, x_i) = 1$ if the output of f changes when an input x_i in p changes; otherwise, $flip(p, x_i) = 0$.

Definition 2: The Relevance Index of an input x_i, denoted by $RI(x_i)$, is defined as

$$RI(x_i) = \frac{\sum_{j=1}^{k} flip(p_j, x_i)}{k} \quad (1)$$

where p_j is the j^{th} simulated pattern, and k is the total number of simulated patterns.

For example, in Fig. 3(a), we can see that there are 8 simulated patterns. For this small example, the simulated patterns are presented in a truth table. The output of f changes

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)

b	c	a	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(b)

Fig. 3. The truth table of $f = ab + ac$ (a) with the input order $a > b > c$. (b) with the input order $b > c > a$.

	n inputs
Base pattern	1011 ··· 011
Bit-flip pattern$_0$	0011 ··· 011
Bit-flip pattern$_1$	1111 ··· 011
Bit-flip pattern$_2$	1001 ··· 011
⋮	⋮
Bit-flip pattern$_{n-1}$	1011 ··· 010

Fig. 4. A base pattern and its corresponding bit-flip patterns for an n-input function.

when the input c changes in two of these patterns. Therefore, $RI(c) = \frac{2}{8} = 0.25$. On the other hand, the output of f changes when the input a changes in six of the patterns, as shown in Fig. 3(b). As a result, $RI(a) = \frac{6}{8} = 0.75$. According to the results, we can see that the input a has a higher RI than the input c in the function $f = ab + ac$. In fact, RI reveals the behavior of inputs x_i in a function. Once an input has a higher RI, the change of this input is more likely to affect the output value of a function.

Since it is intractable to enumerate the whole truth table for evaluating RI in high dimension Boolean space, we adopt an alternative approach, as shown in Fig. 4, to obtain the RI of each input. First of all, we randomly select a base pattern from the given n-input simulation model. To observe the RI of each input, we generate a set of bit-flip patterns, which have the hamming distance of one with respect to the base pattern. Note that the user can determine the number of base patterns according to the number of inputs in the function. For an input x_i with $RI(x_i) = 0$, i.e., its change most likely cannot affect the output value of a function, it is considered as a dummy input and will be removed. In addition to these inputs with $RI(x_i) = 0$, we can also remove the inputs which have a small value of RI when the number of inputs is very large. By adjusting the number of removed inputs carefully, we can obtain a qualified result within the CPU time limit.

C. Expansion order

After having the input patterns in the minimization procedure, the next step is to minimize the circuit with respect to this set of patterns. Basically, we can exploit the SAT-based minimization procedure mentioned in Section II.C to obtain the circuit. However, we found that it is quite difficult to obtain the minimized SOP for the functions in high dimension space by the work of [13]. Therefore, we propose a light-weight algorithm to determine the input expansion order such that the *reduce* operation or the recursive process in the minimization procedure

a	b	c	d	f
0	0	0	0	
0	0	0	1	
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	0
1	0	1	0	0
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	1

(a)

	x_i			
	a	*b*	*c*	*d*
f(on, x_i=1)	1	2	2	1
f(on, x_i=0)	2	1	1	2
f(off, x_i=1)	2	0	2	2
f(off, x_i=0)	1	3	1	1

(b)

Pull(on, x_i)	2	5	3	2
Pull(off, x_i)	4	1	3	4

(c)

Diff(x_i)	-2	4	0	-2

(d)

Priority(1111, x_i)	2	-4	0	2
Priority(0100, x_i)	-2	-4	0	-2
Priority(0010, x_i)	-2	4	0	-2

(e)

Fig. 5. (a) The truth table to be minimized under partial simulations. (b)∼(e) The processes in determining the expansion order of the minimization procedure.

can be skipped. Here we use a function in Fig. 5 as an example to demonstrate the proposed algorithm.

Definition 3: The number of minterms in the on-set of the given function f for which inputs $x_i = 1$ and $x_i = 0$ are denoted as $f(on, x_i = 1)$ and $f(on, x_i = 0)$, respectively. Similarly, the number of minterms in the off-set of the given function f for which inputs $x_i = 1$ and $x_i = 0$ are denoted as $f(off, x_i = 1)$ and $f(off, x_i = 0)$, respectively.

In fact, the expansion of a minterm means to include other minterms in the on-set and then grow as a larger cube. As a result, the idea behinds the proposed algorithm is to determine an expansion order which can maximize the number of included minterms in the on-set after the expansion. In this way, a minterm in the on-set is less likely to be blocked by the minterms in the off-set under the expansion order. Therefore, in this algorithm, we calculate the number of minterms in the on-set and off-set for which inputs $x_i = 1$ and $x_i = 0$ first.

Fig. 5(a) is the truth table of the function in Fig. 1(a). Both on-set and off-set have three minterms after the simulation of selected patterns. Fig. 5(b) lists the results of $f(on, x_i = 1)$, $f(on, x_i = 0)$, $f(off, x_i = 1)$, and $f(off, x_i = 0)$ based on Fig. 5(a). For example, the function has three minterms, 0010, 0100, and 1111, in the on-set. By Definition 3, we know that $f(on, a = 1) = 1$ and $f(on, a = 0) = 2$ because there are 1 and 2 minterms in the on-set for which input $a = 1$ and $a = 0$, respectively. Similarly, we can obtain $f(off, a = 1) = 2$ and $f(off, a = 0) = 1$ as well, as shown in the second column of Fig. 5(b).

Definition 4: $Pull(on, x_i)$ and $Pull(off, x_i)$ are defined as follows:

$$Pull(on, x_i) = f(on, x_i = 1) + f(off, x_i = 0) \quad (2)$$

$$Pull(off, x_i) = f(on, x_i = 0) + f(off, x_i = 1) \quad (3)$$

We calculate $Pull(on, x_i)$ and $Pull(off, x_i)$ by EQ(2) and EQ(3). For example, $Pull(on, a) = f(on, a = 1) + f(off, a = 0) = 1 + 1 = 2$, and $Pull(off, a) = f(on, a = 0) + f(off, a = 1) = 2 + 2 = 4$. The results of this example can be found

in Fig. 5(c). The input variable x_i with a larger $Pull(on, x_i)$ means that the input variable is more likely to be 1 in the on-set. Therefore, expanding this input variable to 1 is *less likely* to overlap the minterms in the off-set as compared with the other input variables. Similarly, the input variable x_i with a larger $Pull(off, x_i)$ means that the input variable is *more likely* to be 0 in the on-set. As a result, expanding this input variable to 1 is more likely to overlap the minterms in the off-set as compared with the other input variables.

Definition 5: $Diff(x_i)$ is defined as follows:

$$Diff(x_i) = Pull(on, x_i) - Pull(off, x_i) \tag{4}$$

We calculate $Diff(x_i)$ by EQ(4). Since $Pull(on, x_i)$ and $Pull(off, x_i)$ represent the opposite information in the algorithm, these two parameters can be merged into a single parameter for further evaluation. Thus, we use EQ(4) to obtain an order of these input variables. In summary, the input variable x_i with a larger $Diff(on, x_i)$ means that the input variable is more likely to be 1 or less likely to be 0 in the on-set.

Definition 6: $Priority(m, x_i)$ is defined as follows:

$$Priority(m, x_i) = \begin{cases} -Diff(x_i), & \text{if } (x_i = 1) \text{ in } m \\ Diff(x_i), & \text{if } (x_i = 0) \text{ in } m \end{cases} \tag{5}$$

where m is a minterm to be expanded.

Now, we can determine the expansion order for a selected minterm by sorting the $priority(m, x_i)$ in EQ(5) in a descending order. If the input variable x_i of 1 in a cube is going to be expanded, the expansion will include all the corresponding minterms with x_i of 0. Since an input variable x_i with a larger $Diff(on, x_i)$ means that the input variable is more likely to be 1 in the on-set, including the minterms with x_i of 0 would not successfully grow as a larger cube with a high probability. Therefore, the priority of this input variable needs to be adjusted by applying a negation on $Diff(x_i)$. The results are shown in Fig. 5(e).

For example, the priority values of input variables (a, b, c, d) under the minterm 1111 are $(2, -4, 0, 2)$. Therefore, the expansion order for minterm 1111 can be $a > d > c > b$ or $d > a > c > b$. Expanding this minterm in one of these two orders can obtain the prime implicant as shown in Fig. 1(b). On the contrary, expanding from the least priority value, which starts from b, will result in a smaller prime implicant as shown in Fig. 1(c). Similarly, expanding another minterm 0100 with the computed expansion order $c > a > d > b$ or $c > d > a > b$ can also obtain the largest prime implicant, which contains 8 minterms. In this example, by using the computed order for expansion on the minterms, an optimized SOP can be obtained.

D. Fixing circuit

Since we do not simulate all the patterns from the simulation model due to the large input space, the missed patterns are temporarily considered as don't care in the minimization procedure. However, the missed patterns might not be don't care patterns actually. Thus, after obtaining the synthesized circuit, we can fix the circuit if we have not reached the CPU time limit in the minimization procedure. That is, we can simulate additional patterns on both the synthesized circuit and the simulation model for comparing and fixing the functional difference if necessary.

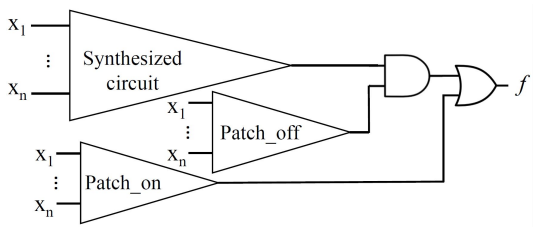


Fig. 6. The overall structure of fixing circuit.

Specifically, we collect the patterns having the output of 1 in the synthesized circuit but having the output of 0 in the simulation model, which form $patch\_off$, and connect $patch\_off$ and the synthesized circuit with an AND gate. This circuit correction will remove the wrong minterms that should have the output of 0 from the synthesized circuit. Similarly, we collect the patterns having the output of 0 in the synthesized circuit but having the output of 1 in the simulation model to form $patch\_on$. We connect $patch\_on$ and the modified circuit we have obtained in the previous step with an OR gate. This circuit correction will include the right minterms that should have the output of 1 into the modified circuit. The overall structure of fixing circuit is as shown in Fig. 6.

IV. Application and Experimental results

In this section, we show the application in ECO flow of the proposed algorithm and the corresponding experimental results. We implemented the proposed algorithm in C++ language. The experiments were conducted on a 2.6 GHz Linux platform (CentOS 6.7). The benchmarks used in the experiments are provided by the contributor of Problem A in the CAD Contest@ICCAD 2019 and are available online [25]. These benchmarks consist of IWLS 2005 benchmarks [26], OpenCore [27], and some industrial designs. Note that we will apply the existing optimization tool, ABC [21], to further optimize the synthesized circuit in the end for exploiting logic sharing among several output functions since the benchmarks have been decomposed into many single-output functions

A. Application – ECO flow

The ECO problem is the task of incrementally updating an implementation of the design when its specification has been changed. The ECO flow consists of the following three steps. First, identify the differences between the old circuit and the new (golden) one. Second, select a set of internal wires as the input of the patch function. Third, synthesize the patch function with the selected internal wires. The proposed algorithm for determining the expansion order can be seamlessly integrated into the third step of the existing ECO optimization flow.

Most of time, the chip designers adopt the ECO flow due to the tight chip release schedule. Therefore, an efficient optimization algorithm is essential to the ECO flow. Besides, chips only reserve a limited amount of spare cells for synthesizing an ECO patch. Hence, designers hope that the size of synthesized patch function can be as minimal as possible. In this experiment, we compare our approach with the re-implemented recursive algorithm [13], and demonstrate the efficiency and effectiveness of the proposed expansion order algorithm in the minimization procedure.

TABLE I
THE COMPARISON OF CPU TIME AND CUBE COUNT BETWEEN THE RE-IMPLEMENTED WORK OF [13] AND THE PROPOSED ALGORITHM.

			Re-implemented [13]		Ours			
	|PI|	|PO|	Time (s)	|SOP|	Time (s)	|SOP|	Speedup	|SOP| diff.
case 1	121	38	2.6	255	0.6	255	4.33	0
case 2	53	19	>1 day	-	1214.0	384479	-	-
case 3	72	1	<0.1	0	<0.1	0	-	-
case 4	56	5	67665.7	193	386.3	201	175.16	8
case 5	87	16	37422.3	698	96.7	710	386.99	12
case 6	76	1	14314.0	1023	22.7	1023	630.57	0
case 7	43	7	0.3	27	0.1	28	3.00	1
case 8	44	5	>1 day	-	81.0	1532	-	-
case 9	173	16	23254.2	2134	239.2	2130	97.22	-4
case 10	37	2	58.1	128	1.2	128	48.42	0
case 11	60	20	>1 day	-	876.0	18910	-	-
case 12	40	26	>1 day	-	72.6	13060	-	-
case 13	43	7	0.3	16	0.1	17	3.00	1
case 14	50	22	>1 day	-	542.6	53410	-	-
case 15	80	3	>1 day	-	8.3	1279	-	-
case 16	26	4	1.0	78	0.1	78	10.00	0
case 17	76	33	9587.7	150	164.5	149	58.28	-1
case 18	102	2	>1 day	-	5853.3	261271	-	-
case 19	73	8	1247.7	17	169.9	18	7.34	1
case 20	51	2	80629.3	384	357.6	384	225.47	0

Since the benchmarks have dozens of inputs, we cannot simulate all the input patterns efficiently. Therefore, we conducted functional pruning on these benchmarks. That is, we first removed the inputs with small values of RI for the minimization procedure. Then, we simulate the whole truth table for both approaches.

TABLE I shows the experimental results of the first application. Column 1 lists the benchmarks. Columns 2 and 3 list the numbers of primary inputs and primary outputs of these benchmarks. Columns 4 and 5 show the CPU time and the number of cubes in the resultant SOP form using the re-implemented recursive algorithm [13]. Columns 6 and 7 show the corresponding results by using our approach. Column 8 shows the speedup for our approach as compared to the re-implemented [13]. The last column shows the difference of cube count between two approaches.

For example, considering case 6 benchmark, the re-implemented recursive algorithm [13] cost 14314 seconds (≈ 4 hours) to synthesize the patch function while our approach only cost 22.7 seconds (0.5 minute). Furthermore, the cube count in the synthesized SOP form for both approaches are the same, 1023.

According to the experimental results, we also observe that the recursive algorithm cannot finish cases 2, 8, 11, 12, 14, 15, and 18 (7 out of 20 or 35% benchmarks) in 86,400 seconds while ours can. The average CPU time for all the benchmarks in our approach is only around 500 seconds. For the benchmarks that have results in the both approaches, the ratio of speedup of the proposed expansion order algorithm with the minimization procedure is up to 630. Furthermore, the number of cubes in the resultant SOP of our algorithm is quite close to the work of [13] for all the benchmarks with results. For case 9 benchmark, the cube count of synthesized circuit is even lower by using our approach. Note that the number of cubes in the resultant SOP of case 3 is zero because the results of all the simulated patterns are 1 in this case. In other words, the functionality of case 3 is very close to a constant of 1. According to these experimental results, we know that the proposed algorithm is very efficient and can be well integrated with the ECO flow in high quality.

B. *Influence of simulated patterns*

For the designs in high dimensions, the number of simulated patterns definitely influences the accuracy of the synthesized circuit. However, we cannot conduct extraordinary amount of simulations on the simulation model due to the CPU time constraint. Thus, we need to utilize the simulation in a more effective way for obtaining higher accuracies.

Here, we conduct another experiment to observe the relationship between the number of selected inputs and accuracy under the same amount of simulated patterns, $2^{20} = 1,048,576$ patterns. In Section II.B, we have mentioned that we can determine the number of selected inputs by the magnitude of RI. The size of synthesized circuit is measured by two-input gates. The accuracy is obtained by conducting simulation of 100,000 random patterns on the synthesized circuit and the simulation model at the same time. Note that one correct matching means all output values of the synthesized circuit must be the same as the output generated by the simulation model under an input pattern.

TABLE II shows the experimental results of the second experiment. Column 1 lists the benchmarks. Columns 2, 3, and 4 show the accuracy, CPU time, and gate count of the synthesized circuit with at most 20 selected inputs. Columns 5, 6, and 7 show the corresponding results with at most 21 selected inputs. Columns 8, 9, and 10 show the corresponding results with at most 22 selected inputs. According to TABLE II, we first observe that the accuracy is decreased when the number of selected inputs increases for most of the benchmarks. This is because the local input space is double when we increase the number of selected inputs by 1. Second, we do not see much difference in CPU time except for the benchmarks case 2 and case 14. Generally, the CPU time is proportional to the number of simulated patterns in the minimization procedure. In this experiment, we adopted $2^{20} = 1048576$ randomly simulated patterns for all the different numbers of selected inputs. However, these two benchmarks are special functions,

TABLE II
THE EXPERIMENTAL RESULTS FOR DIFFERENT NUMBERS OF SELECTED INPUTS UNDER THE SAME AMOUNT OF SIMULATION PATTERNS.

	$\|\text{Selected inputs}\| \leq 20$			$\|\text{Selected inputs}\| \leq 21$			$\|\text{Selected inputs}\| \leq 22$		
	Accuracy (%)	Time (s)	Gate count	Accuracy (%)	Time (s)	Gate count	Accuracy (%)	Time (s)	Gate count
case 1	100	7.5	170	100	7.9	170	100	6.6	170
case 2	87.79	1271.0	27227	18.08	3220.4	1417647	5.62	5805.8	2242515
case 3	100	1.2	0	100	4.6	0	100	9.8	0
case 4	99.48	123.8	110	99.21	92.1	4922	96.08	135.7	37736
case 5	99.26	25.8	137	99.24	25.7	137	99.23	25.9	137
case 6	99.97	76.0	43	98.16	82.6	17470	95.04	129.6	42053
case 7	100	1.8	40	100	1.9	40	100	2.0	40
case 8	99.98	46.3	36	98.96	39.1	10133	98.22	46.4	17234
case 9	69.22	2739.4	1317	65.03	2186.9	90180	58.76	2844.2	274136
case 10	100	7.2	24	100	6.3	24	100	6.5	24
case 11	99.39	723.3	205	92.47	663.9	74279	86.93	885.3	132817
case 12	99.72	226.0	192	89.18	299.3	97335	77.61	498.6	203499
case 13	100	1.7	27	100	2.1	27	100	2.3	27
case 14	41.67	1045.1	69158	21.41	2058.2	933135	7.69	4693.2	2032481
case 15	99.47	167.8	83	95.47	163.1	38280	90.74	229.1	78753
case 16	100	1.2	34	100	1.3	34	100	1.2	34
case 17	99.52	121.4	111	99.59	91.9	766	99.12	97.3	4291
case 18	54.03	424.2	61249	52.86	426.2	222380	52.2	686.6	318468
case 19	98.63	181.1	37	98.57	119.7	100	98.79	138.0	189
case 20	72.07	68.9	27	77.73	59.3	4095	77.93	96.6	13217

which act as a parity function by our examination. Since the output of a parity function is 1 if and only if the input pattern has an odd number of ones, we cannot find many adjacent minterms in on-set for minimization. In other words, a minterm in the on-set is easily to overlap the minterms in the off-set when expanding. Since this situation occurs frequently when the number of selected inputs was increased, the CPU time has also increased. Finally, we also see a tremendous increase about the number of gate counts for most of the benchmarks. This is because the synthesized circuits become more complicated as the number of selected inputs was increased.

In summary, we conclude that under the constraint of simulating the same amount of patterns, determining an appropriate amount of selected inputs first and then enumerating the truth table for these selected inputs is a better approach to the minimization problem in high dimension space. Note that since the input dimension of each single-output function of cases 1, 3, 7, 10, 13, and 16 is less than 20, we can enumerate the complete truth tables for these cases such that the accuracies are all 100% under different numbers of selected inputs.

V. CONCLUSION

In this paper, we propose an optimization framework, which consists of functional pruning, SAT-based minimization with expansion order algorithm, and fixing network. According to experimental results, the proposed expansion order algorithm improved the bottleneck in the previous work and completed the optimization flow for high dimension Boolean space.

REFERENCES

[1] R. J. Bayardo Jr and R. C. Schrag, "Using CSP look-back techniques to solve real world SAT instances," in *Proc. AAAI*, pp. 203-208, 1997.

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

[3] Y.-C. Chen and C.-Y. Wang, "Fast detection of node mergers using logic implications," in *Proc. ICCAD*, pp. 785-788, 2009.

[4] O. Coudert, J. Madre, and H. Fraisse, "A new viewpoint on two-level logic minimization," in *Proc. DAC*, pp. 625-630, June 1993.

[5] E. L. Lawler, "An approach to multilevel boolean minimization," *Journal of the ACM*, 1964.

[6] A. Mishchenko, R. K. Brayton, A. Petkovska, M. Soeken, L. Amaru, and A. Domic, "Canonical computation without canonical representation," in *Proc. DAC*, pp. 1-6, 2018.

[7] J. P. Marques-Silva and K. A. Sakallah, "GRASP-A new search algorithm for satisfiability," in *Proc. ICCAD*, pp. 220-227, 1996.

[8] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. on Computers*, pp. 506-521, 1999.

[9] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. tech J.*, vol. 35, no. 5, pp. 1417-1444, Nov. 1956.

[10] P. McGeer, J. Sanghavi, and R. K. Brayton, "Espresso-signature: A new exact minimizer for logic functions," *IEEE Trans. on Very Large Scale Integration Systems*, pp. 618-624, 1993.

[11] C. D. Murray, and R. R. Williams, "On the (non) NP-hardness of computing circuit complexity," *Theory of Computer*, 2017.

[12] W. Quine, "The problem of simplifying truth functions," *American Mathematical Monthly*, vol. 59, no. 8, pp. 521-531, 1952.

[13] D. A. Quoc, M. P.-H. Lin, N.-Z. Lee, L.-C. Chen, J.-H. R. Jiang, A. Mishchenko, and R. K. Brayton, "Efficient computation of ECO patch functions," in *Proc. DAC*, pp. 1-6, 2018.

[14] H. Savoj, "Improvements in technology independent optimization of logic circuits," in *Proc. IWLS*, pp. 1-6, 1997.

[15] H. Savoj and R. K. Brayton, "The use of observability and external don't-care for the simplification of multi-level networks," in *Proc. DAC*, pp. 291-301, 1990.

[16] K.-F. Tang, P.-K. Huang, C.-N. Chou, and C.-Y. Huang, "Multi-patch generation for multi-error logic rectification by interpolation with cofactor reduction," in *Proc. DATE*, pp. 1567-1572, 2012.

[17] G. Tseitin, "On the complexity of derivation in propositional calculus," Studies in constructive mathematics and mathematical logic, vol. 2, no. 115-125, pp. 10–13, 1968.

[18] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Trans. on Computer-Aided Design*, pp. 1230-1246, 2006.

[19] B.-H. Wu, C.-J. Yang, C.-Y. Huang, and J.-H. R. Jiang, "A robust functional ECO engine by SAT proof minimization and interpolation techniques," in *Proc. ICCAD*, pp. 729-734, 2010.

[20] H. Zhang and J.-H. R. Jiang, "Cost-aware patch generation for multi-target function rectification of engineering change orders," in *Proc. DAC*, pp. 1-6, 2018.

[21] Berkeley Logic Synthesis and Verification Group, "ABC: a system for sequential synthesis and verification," Available: https://people.eecs.berkeley.edu/∼alanmi/abc/.

[22] http://minisat.se/.

[23] https://www.labri.fr/perso/lsimon/glucose/.

[24] https://github.com/arminbiere/cadical.

[25] http://iccad-contest.org/2019/problems.html.

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

[27] http://opencores.org/.