Fast Adjustable NPN Classification Using Generalized Symmetries

Xuegong Zhou, Lingli Wang
State Key Laboratory of ASIC and System
Fudan University, Shanghai, China
{zhouxg, llwang}@fudan.edu.cn

Peiyi Zhao
Integrated Circuit and Embedded Systems Lab
Chapman University, Orange, CA, USA
zhao@chapman.edu

Alan Mishchenko
Department of EECS
University of California at Berkeley, Berkeley, CA, USA
alanmi@berkeley.edu

***Abstract*—NPN classification of Boolean functions is a powerful technique used in many logic synthesis and technology mapping tools in FPGA design flows. Computing the canonical form of a function is the most common approach of Boolean function classification. In this paper, a novel algorithm for computing NPN canonical form is proposed. By exploiting symmetries under different phase assignments and higher-order symmetries of Boolean functions, the search space of NPN canonical form computation is pruned and the runtime is dramatically reduced. The algorithm can be adjusted to be a slow exact algorithm or a fast heuristic algorithm with lower quality. For exact classification, the proposed algorithm achieves a 30× speedup compared to a state-of-the-art algorithm. For heuristic classification, the proposed algorithm has similar performance as the state-of-the-art algorithm with a possibility to trade runtime for quality.**

Keywords—NPN classification; Boolean matching; symmetry; canonical form; cofactor signature

I. Introduction

Classification of Boolean functions is the task of grouping similar functions into equivalence classes. A related problem is Boolean matching, which checks whether two functions belong to the same equivalent class.

The most frequently used classification method is based on Negation-Permutation-Negation (NPN) equivalence. Two single-output Boolean functions are NPN equivalent, if one of them can be obtained from the other by negating inputs, permuting inputs, and negating the output.

NPN classification has many applications in logic synthesis [1]-[4] and technology mapping [5]-[9] for FPGAs. In synthesis, optimal or near-optimal circuits for a large number of practical functions can be precomputed and stored in a library. With Boolean function classification, only one function in each equivalence class needs to be in the library, resulting in a dramatic reduction of the library size [2][4].

In technology mapping, NPN classes of functions that can be matched against a programmable cell are pre-computed and stored in a hash-table, to allow for a quick constant-time check whether a function is realizable using the given cell. Several recent studies are based on this approach [8][9].

In these applications, the speed of NPN classification determines the speed of the synthesis engine or the technology mapper. This is because library precomputation can be done offline, but NPN classification is done online during runtime. Similarly, if the quality of NPN classification is poor, more precomputation has to be done and the resulting library takes more memory.

A common approach of NPN classification is to construct a canonical form for a Boolean function f, and use this canonical form as the representative of the equivalence class f belongs to. There are several NPN canonical forms, such as the function with the smallest truth table representation [10][11], or with the smallest spectrum representation in the NPN equivalence class [12]-[14].

Naïve computation of the canonical form requires exhaustive enumeration of all the possible transformations. For NPN classification of n input Boolean functions, $n!$ permutations and 2^{n+1} negations should be enumerated. Various methods based on signatures [10][11] and variable symmetry [10][11][13]-[16] are used to reduce the computation cost. However, none of the method works well for *all* functions. In some cases, semi-canonical form can be used instead of the exact canonical form [15][17] in order to get practical runtime.

The main contributions of the paper are:

- An in-depth study of NPN classification as a theoretical guidance to define new canonical forms, and to determine how the canonical form computation can be adapted in a given application.
- A hybrid NPN canonical form introduced by combining the cofactor signature and the truth table. By exploiting various symmetries of Boolean functions, this canonical form is computed efficiently for many Boolean functions.
- A new heuristic method is introduced to classify rare difficult functions without symmetric variables, whose input variables cannot be distinguished by the signatures. By adjusting the exact-to-heuristic ratio, the proposed algorithm trades runtime for the classification quality.

The proposed approach is the most general among the existing ones while at the same time being the most practical. The proposed implementation in ABC outperforms other methods in each category. In particular, the exact method is 30x faster than the best available exact method, while the heuristic method has similar runtime but better quality, and allows for a number of other quality/runtime tradeoffs.

The paper is organized as follows. Section II formally introduces the terminology. Section III defines the generalized variable symmetry. Section IV defines the hybrid canonical form. Section V describes the algorithm. Section VI shows the results of the experimental evaluation, and Section VII concludes the paper.

II. Preliminaries

A. Boolean Function

This paper deals with completely specified Boolean functions, $f(X): B^n \to B, B = \{0,1\}$, where $X = (x_1, x_2, \dots x_n)$ is a bit vector of size n, $x_i \in B$. When the input bit vector is considered as a binary number, whose value is m, the corresponding bit vector is denoted as $X_{(m)}$. The *truth table* of function f is a bit vector of size 2^n, composed of the output value of the function: $T(f) = (f(X_{(2^n-1)}), \dots, f(X_{(1)}), f(X_{(0)}))$.

A *literal* $\dot{x}$ is a variable x or its complement $\bar{x}$. A *cube* is the Boolean conjunction of literals. A *minterm* is a cube with n literals. The *satisfy count* of a function, denoted as $|f|$, is the number of on-set minterms covered by f.

The *cofactor* of f with respect to a literal $\dot{x}$, denoted as $f_{\dot{x}}$, is the function obtained by setting $\dot{x}$ to 1 in f. The cofactor of f with respect to cube c, denoted as f_c, is the function obtained by setting all literals of the cube to 1.

A Boolean function f is called *balanced* if $|f| = |\bar{f}|$. An input variable x is called *balanced* if $|f_x| = |f_{\bar{x}}|$.

B. NPN Equivalence

Definition 1: An *NPN transformation* on a Boolean function is a phase assignment, followed by a permutation of its input variables, followed by a polarity assignment of its output. Applying transformation τ to function f is denoted as $f \circ \tau$.

For a Boolean function of n inputs, there are $2^{n+1}n!$ distinct NPN transformations. We denote a transformation τ by a vector of literals to indicate the permutation and phases of the inputs and a z indicating the polarity of the output. For example, applying transformation $\tau = (\overline{x_2}, x_1, x_3, \bar{z})$ to function $f(x_1, x_2, x_3) = x_1x_2 + x_2x_3$ results in a new function $f \circ \tau = \overline{\overline{x_2}x_1 + x_1x_3}$, which replaces x_1 with $\overline{x_2}$, and x_2 with x_1 respectively, while also negating the output.

Definition 2: Two Boolean functions, f and g, are *NPN equivalent*, denoted as $f \equiv g$, if there exists an NPN transformation τ, such that $f \circ \tau$ is equal to g.

The NPN equivalence is an equivalence relation, which partitions all single-output Boolean functions into equivalence classes. The *NPN equivalence class* of a function f is denoted $[f]$ and is defined as the set of functions that are NPN equivalent to f, i.e. $[f] = \{g \mid f \equiv g\}$.

As an example, for function $f(x_1, x_2, x_3) = x_1x_2 + x_3$, its NPN equivalence class contains 48 functions, such as $x_1\overline{x_3} + x_2$ and $\overline{x_2x_3 + \overline{x_1}}$, with their corresponding NPN transformations $(x_1, \overline{x_3}, x_2, z)$ and $(x_2, x_3, \overline{x_1}, \bar{z})$. There are in total $2^{3+1}3!$ =96 different NPN transformations of 3 variable functions. Some transformations may produce the same function because of the variable symmetry, hence the equivalence class contains much less functions than 96.

III. Symmetry Relationship

A. First-Order Symmetries

Definition 3 (variable symmetry): Two variables x_i and x_j are said to be symmetric in function f, which is denoted by $x_i \leftrightarrow x_j$, if f is invariant under an exchange of x_i and x_j, i.e. $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, x_j, \dots, x_i, \dots)$. This classical symmetry is called *nonequivalent-symmetry (NE-symmetry)*. When considering the phase assignment, if f is invariant under an exchange of x_i and $\bar{x}_j$, i.e. $f(\dots, x_i, \dots, \bar{x}_j, \dots) = f(\dots, \bar{x}_j, \dots, x_i, \dots)$, variables x_i and x_j are said to be *equivalent-symmetric (E-symmetric)*, which is denoted by $x_i \leftrightarrow \bar{x}_j$. If x_i and x_j are simultaneously NE- and E-symmetric, then x_i and x_j are said to be *multiform symmetric*, which is denoted by $x_i \overset{m}{\leftrightarrow} x_j$.

It can be shown using Boole's expansion theorem that variable symmetry is equivalent to the equality of the cofactor pair, i.e. $x_i \leftrightarrow x_j$ if and only if $f_{\bar{x}_i x_j} = f_{x_i \bar{x}_j}$; $x_i \leftrightarrow \bar{x}_j$ if and only if $f_{x_i x_j} = f_{\bar{x}_i \bar{x}_j}$[18]. There are a number of efficient algorithms in the literatures for symmetry detection [18][19].

By negating one symmetric variable of the function f, the E-symmetry is converted to NE-symmetry. Therefore, in the classification process, only NE-symmetry and multiform symmetry are considered. The multiform symmetry cannot be simply regarded as NE-symmetry, and must be manipulated separately. Many existing classification methods [13][15] neglected this difference.

The NE-symmetry and the multiform symmetry are both equivalence relations, while the E-symmetry is not. Hence, these equivalence relations can be used to partition the input variables $x_1, x_2, \dots x_n$ into equivalence classes.

Definition 4 (symmetric class): When the variables in an equivalence class are NE-symmetric, this class is called a *NE-symmetric class*, which is denoted by $[x_{i_1}, x_{i_2}, \dots x_{i_m}]$. When the variables in an equivalence class are multiform symmetric, this class is called a *multiform symmetric class*, which is denoted by $\langle x_{i_1}, x_{i_2}, \dots x_{i_m} \rangle$.

The phases of variables in a NE-symmetric classes are determined, while the phases of variables in a multiform classes are undetermined, an arbitrary phase can be chosen for these variables.

The following theorem is useful for manipulating the phase assignment of the multiform symmetric classes.

Lemma 1: Let x_i and x_j be E-symmetric, $x_i \leftrightarrow \bar{x}_j$, then $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, \bar{x}_j, \dots, \bar{x}_i, \dots)$.

Proof: $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, x_i, \dots, \bar{\bar{x}}_j, \dots) = f(\dots, \bar{x}_j, \dots, \bar{x}_i, \dots)$. ■

Theorem 1: Let x_i and x_j be multiform symmetric, $x_i \overset{m}{\leftrightarrow} x_j$, then $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, \bar{x}_i, \dots, \bar{x}_j, \dots)$.

Proof: $f(\dots, x_i, \dots, x_j, \dots) = f(\dots, x_j, \dots, x_i, \dots) = f(\dots, \bar{x}_i, \dots, \bar{x}_j, \dots)$. ■

Theorem 1 indicates that, when negating even number of the variables in a multiform symmetric class, the function is invariant. Only two different phase assignments of an m variable multiform symmetric class need to be considered, instead of the whole 2^m phase assignments.

B. Higher-Order Symmetries

The symmetric relation of two variables can be extended to two symmetric classes, as a second-order symmetry [20].

Definition 5 (second-order symmetry): Two NE-symmetric classes or two multiform symmetric classes with the same size $C_i = (x_{i_1}, x_{i_2}, \dots x_{i_m})$ and $C_j = (x_{j_1}, x_{j_2}, \dots x_{j_m})$ of a function f are said to be *NE-symmetric*, if f is invariant under an exchange of C_i and C_j, i.e. $f(\dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots, x_{j_1}, \dots, x_{j_2}, \dots, x_{j_m}, \dots) = f(\dots, x_{j_1}, \dots, x_{j_2}, \dots, x_{j_m}, \dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots)$; C_i and C_j are said to be *E-symmetric*, if f is invariant under an exchange of C_i and $\overline{C_j}$, i.e. $f(\dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots, \bar{x}_{j_1}, \dots, \bar{x}_{j_2}, \dots, \bar{x}_{j_m}, \dots) = f(\dots, \bar{x}_{j_1}, \dots, \bar{x}_{j_2}, \dots, \bar{x}_{j_m}, \dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots)$. If C_i and C_j are simultaneously NE- and E-symmetric, then C_i and C_j are said to be *multiform symmetric*.

Definition 6 (symmetry with single negation): Two multiform symmetric classes with the same size $C_i = \langle x_{i_1}, x_{i_2}, \dots x_{i_m} \rangle$ and $C_j = \langle x_{j_1}, x_{j_2}, \dots x_{j_m} \rangle$ of a function f are said to be *symmetric with single negation (SN-symmetric)*, if f is invariant under an exchange of C_i and C_j with negating a single variable, i.e. $f(\dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots, \bar{x}_{j_1}, \dots, x_{j_2}, \dots, x_{j_m}, \dots) = f(\dots, \bar{x}_{j_1}, \dots, x_{j_2}, \dots, x_{j_m}, \dots, x_{i_1}, \dots, x_{i_2}, \dots, x_{i_m}, \dots)$.

Note that, in the definition of second-order E-symmetry, all the variables in class C_j need to be negated. While in the definition of second-order SN-symmetry, only one variable in class C_j need to be negated. Theorem 1 only valid for first-order multiform symmetry, but not valid for second-order multiform symmetry.

Second-order NE-symmetry and multiform symmetry are both equivalence relations, while second-order E-symmetry and SN-symmetry are not. If two symmetric classes C_i and C_j are NE-symmetric or multiform symmetric, they can be merged into one second-order symmetric class. If they are E-symmetric or SN-symmetric, all the variables ore one of the variable in C_j can be negated to convert the symmetric relation to NE-symmetry, then the two classes can be merged. Three or more symmetric classes can be merged in the same way. This merge process can be operated recursively to generate higher-order symmetric classes.

For example, given a function $f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 x_2 + x_3 \overline{x_4})(x_5 \oplus x_6)$, we have $x_1 \leftrightarrow x_2$, $x_3 \leftrightarrow \overline{x_4}$ and $x_5 \leftrightarrow x_6$. The input variables can be divided into three symmetry classes: $C_1 = [x_1,\ x_2]$, $C_2 = [x_3,\ \overline{x_4}]$ and $C_3 = \langle x_5,\ x_6 \rangle$. Because $C_1 \leftrightarrow C_2$, these two class can be merged into a second-order symmetry class $C_{12} = [[x_1,\ x_2], [x_3,\ \overline{x_4}]]$.

IV. CANONICAL FORM

A. General Definition

The canonical form of a function f is a representative selected among functions of its NPN equivalence class $[f]$ based on a criterion. There are several different canonical forms defined by the previous works [10][13][14][21]. A more general definition is given below.

Definition 7: Let F_n be the set of Boolean functions with n input variables. The *NPN canonical form* is a function of Boolean functions that satisfies two conditions,

$\kappa(f): F_n \to F_n$,

1) $\kappa(f) \in [f]$,

2) $\forall g \in [f]: \kappa(g) = \kappa(f)$

A comparable signature of Boolean functions can be used to define an ordering of Boolean functions, and then a concrete canonical form can be defined by that order.

Definition 8 (ordering of Boolean functions): Let the signature s be a one-to-one relationship between F_n and a strict totally ordered co-domain, such as a subset of integer or vector, a strict total order can be defined on F_n based on s: $\forall f, g \in F_n$, $f <_s g$ iff $s(f) < s(g)$.

A strict totally ordered set S has a unique minimum element, denoted as $\min(S)$, i.e. $\min(S) \in S, \forall a \in S$, $\min(S) \le a$.

Theorem 2: When a strict total order is defined on F_n, the minimum element of $[f]$ is a canonical form of f.

Proof: Examine the two conditions of the canonical form:

1) According to Definition 8, $\min([f]) \in [f]$.

2) Since $\forall g \in [f], [g] = [f]$, the following is true: $\kappa(g) = \min([g]) = \min([f]) = \kappa(f)$ ■

Various signatures are used for computing a canonical form, such as the truth table [10][11], the satisfy count of cofactors [13][14], spectral coefficients [12][14], and specific binary cost [21].

B. Cofactor Signature

The cofactor signature composed of the satisfy counts of cofactors is a well-known signature of Boolean functions, which is adopted in several classification algorithms [2][11][13][15]. Reference [13] defines an NPN canonical form based on the cofactor signature.

The cofactor signature is closely related to the truth table. Several heuristic rules have been proposed to compute the truth table based NPN canonical form, i.e., the function in the NPN equivalence class with the minimum truth table [2][15].

Rule 1 (the output polarity): For a Boolean function f with n input variables, the polarity of the canonical form is determined by the satisfy count of f. If $|f| < |\bar{f}|$, or equally $|f| < 2^{n-1}$, the polarity is positive; if $|f| > |\bar{f}|$, or equally $|f| > 2^{n-1}$, the polarity is negative; if f is balanced, the polarity is not determined.

Rule 2 (the phase of input variables): The phase of each input variable in the canonical form is determined by the satisfy count of the cofactor with respect to that variable. If $|f_{x_i}| < |f_{\overline{x_i}}|$, or equally $|f_{x_i}| < |f|/2$, the phase of x_i is positive; if $|f_{x_i}| > |f_{\overline{x_i}}|$, or equally $|f_{x_i}| > |f|/2$, the phase of x_i is negative; if x_i is balanced, the phase is not determined.

Rule 3 (the ordering of input variables): After the phases of input variables are assigned, their order is determined by the ascending order of the satisfy count of the cofactors with respect to each variable.

These three rules are heuristic, and may not obtain the canonical form with the minimum truth table. However, they can produce the function with the minimum signature vector composed of the satisfy count of the function and the satisfy counts of the cofactors with respect to each variable, i.e. $(|f|, |f_{x_1}|, |f_{x_2}|, \dots, |f_{x_n}|)$. A new canonical form combining the cofactor signatures and the truth table can be defined to make these rules valid.

C. Hybrid Canonical Form

The proposed hybrid canonical form uses the cofactor signature providing the general information about the function, and the truth table providing the exact information.

Definition 9: The *hybrid signature vector* for a function f, denoted by $H(f)$, is a vector composed of the satisfy count of the function and the satisfy counts of the cofactors with respect to each variable, followed by the bits of the truth table,
i.e. $H(f) =$
$(|f|, |f_{x_1}|, |f_{x_2}|, \dots, |f_{x_n}|, f(X_{(2^n-1)}), \dots, f(X_{(1)}), f(X_{(0)}))$.

Theorem 3: For a Boolean function f with n input variables, its hybrid signature vector $H(f)$ uniquely and completely specifies function f.

Proof: The truth table part of the hybrid signature vector uniquely and completely specifies function f, the cofactor signature part can be computed from the truth table. ■

The signature vectors of different functions can be compared via the lexicographic order, which is a strict total order. According to Theorem 2, an NPN canonical form can be defined using $H(f)$.

Definition 10: The hybrid canonical form of a function f is the function in the NPN equivalence class $[f]$ with the minimum hybrid signature vector.

In the hybrid canonical form, the cofactor signature takes precedence over the truth table. It is used to determine the phase and the order of each input variables. Any other signatures that can distinguish input variables, such as the row sums used in [11], can be merged into the canonical form definition. In this paper, only the cofactor signature is used.

D. Variable Symmetry and Canonical Form

As described in Section III, the input variables of a function can be partitioned into symmetric classes. The variables in a symmetric class have the same properties. During the canonical form computation, a symmetric class is handled as a single variable, resulting in the dramatic reduction of the complexity of the algorithm. This method was adopted in several algorithms [13]-[15], but the correctness is not proven.

Actually, this method is invalid for the spectrum based canonical form used in [13] and [14], because it neglects the higher-order coefficients among the variables in a symmetric class. However, the canonical form can be redefined to make this method feasible. The following theorem shows that the variable grouping method can be applied to all the signature based canonical form defined by Theorem 2.

Definition 11: A Boolean function f is *symmetry aggregate* if all of its symmetric variables are placed in adjacent positions within a group in the input variable vector.

For example, function $f_1(x_1, x_2, x_3, x_4, x_5, x_6) = x_1\overline{x_2}x_3 + x_4 + x_5x_6$ is symmetry aggregate, because variables in both symmetric classes$[x_1, x_2, x_3]$ and $[x_5, x_6]$ are placed in adjacent positions in the input variable vector. While $f_2(x_1, x_2, x_3, x_4, x_5, x_6) = x_1\overline{x_2}x_4 + x_3 + x_5x_6$ is not symmetry aggregate, because the variables in symmetric group $[x_1, x_2, x_4]$ are separated in the input variable vector.

Definition 12: The *symmetry aggregate equivalence set* of a Boolean function f, denoted by $[f]^S$, is the set of the symmetry aggregate functions in the NPN equivalence class of f, i.e. $[f]^S = \{h \mid h \in [f], h \text{ is symmetry aggregate }\}$.

Theorem 4: When a strict total order is defined on F_n, the minimum element of $[f]^S$, denoted by $\kappa^S(f)$, is a NPN canonical form of f.

Proof: Examine the two conditions of the canonical form:

1) According to Definition 8 and Definition 12, $\min([f]^S) \in [f]^S \in [f]$.

2) It is true that $\forall g \in [f]$, $[f] = [g]$. According to Definition 12, $[f]^S = [g]^S$, therefore $\kappa^S(g) = \min([g]^S) = \min([f]^S) = \kappa^S(f)$ ■

If the canonical form computation algorithm groups symmetric variables together, the output result of the algorithm is a *symmetry aggregate* function, which is the *symmetry aggregate* canonical form defined in Theorem 4. The proposed algorithm introduced in Section V deal with the symmetry aggregate hybrid canonical form defined via the hybrid signature vector.

Theorem 4 show that the canonical form can be redefined according to the canonical form computing algorithm. This provides more freedom for the algorithm design.

V. Computing the Canonical Form

A. Basic Canonical Form Algorithm

Given a Boolean function f, the proposed algorithm computes an exact NPN canonical form or a semi-canonical form by the hybrid signature vector, according to certain threshold. The algorithm is organized into seven steps, as described below.

Algorithm 1: Compute the NPN canonical form

```
Input: Boolean function $f$ with $n$ input variables
Output: canonical form $\kappa(f)$ or semi-canonical form $\kappa'(f)$
1:  Decide the polarity of the output.
2:  Decide the phases of input variables.
3:  Reorder and group the input variables by the cofactor signature.
4:  Detect variable symmetry and group symmetric variables.
5:  Estimate the cost of the exhaustive enumeration.
6:  If the cost is larger than the enumeration threshold, do the simple enumeration which generates $\kappa'(f)$,
7:  Else do the exhaustive enumeration which generates $\kappa(f)$.
```

In Step 1, the output polarity is determined according to Rule 1 described in Section IV-B. Negate the function if the output polarity is negative. If f is balanced, the polarity is undecided, and the subsequent steps of the algorithm should use both positive and negative polarities, and the resulting function with the smaller truth table is returned.

In Step 2, the input phase assignment of each variable is performed according to Rule 2. Negate the variables whose phase are negative. Recalculate the cofactor signature if necessary.

In Step 3, based on Rule 3, the input variables are reordered such that $|f_{x_1}| \leq |f_{x_2}| \leq \dots \leq |f_{x_n}|$. Then the variables are grouped into groups $G_1, G_2, \dots, G_k$ by the satisfy count of the cofactors with respect to each variable. Variables with the same satisfy count are in the same group. If f contains balanced variables, all of them are grouped in the last group G_k. This group is called a balanced group.

In Step 4, the variables within each group are checked for symmetry, and are divided into symmetric classes. If the group is balanced, the symmetric relation can be NE-symmetry, E-symmetry or multiform symmetry. If E-symmetry is detected, negate one variable to convert the symmetric relation into NE-symmetry. If the group is not balanced, only NE-symmetry need to be checked.

Then higher-order symmetries are detected, and the lower-order symmetric classes are merged into higher-order symmetric classes.

After Step 4, the input variables are grouped into several groups, and each group is divided into several symmetric classes. i.e., $G_1 = \left(C_1, C_2, \ldots, C_{m_1}\right)$, $G_2 = \left(C_{m_1+1}, C_{m_1+2}, \ldots, C_{m_2}\right)$, …, $G_k = \left(C_{m_{k-1}+1}, C_{m_{k-1}+2}, \ldots, C_m\right)$, $1 \le m \le k$, $1 < m_1 < m_2 < \cdots < m_{k-1} < m$. Each symmetric class C_i contains one or more symmetric variable.

In Step 5, three factors are used to estimate the cost of the exhaustive enumeration. $c_p = ln(m_1!\,(m_2 - m_1)! \ldots (m - m_{k-1})!)$, represents the permutation cost; $c_n =$ the number of balanced variables, represents the negation cost; $c_t = n$ (the number of input variables), represents the truth table manipulating cost.

We use the logarithm of the exhaustive enumeration runtime as the enumeration cost, and assume it is a linear combination of $\mathrm{c_p}$, c_n and c_t. We recorded $\mathrm{c_p}$, c_n, c_t and the runtime of each function in the experiment, and computed the coefficients via linear regression.

Step 6 is a fast greedy enumeration method introduced by Huang *et al.* [15]. Adjacent symmetric classes in each group are swapped and flipped. Totally 8 transformations are considered $(ab, a\bar{b}, \bar{a}b, \overline{ab}, ba, b\bar{a}, \bar{b}a, \overline{ba})$ if the group is balanced. Otherwise only two transformations are considered (ab, ba) . The transformation leading to the smallest truth table is chosen. This procedure is repeated until no improvement produced. This step generates a semi-canonical form.

Step 7 exhaustively enumerate all the different transformations, and save the minimum truth table as the canonical form. Totally $m_1!\,(m_2 - m_1)! \ldots (m - m_{k-1})!$ different permutations need to be enumerated. If G_k.is not balanced, the phase of all the variables are determined. Otherwise, $2^{m - m_{k-1}}$ different phase assignment need to be enumerated. This step generates an exact canonical form.

Instead of the recursive enumeration used in many previous algorithms [13]-[15], we use iterative enumeration. By using Johnson-Trotter permutation algorithm [22] and Gray code, each enumeration step only swap one pair of adjacent symmetric classes, or flip one symmetric class. Thus the transformation cost during enumeration is minimized.

The algorithm keeps a record of the first-level multiform symmetric class. When performing phase enumeration, only one variable in the class need to be negated instead of all the variables in the class, according to Theorem 1.

B. *Hierarchical Canonical Form Algorithm*

Petkovska *et al.* [17] introduced a hierarchical method, which reuse the intermediate results to speed up the classification process. This method can be used with the algorithm described in last section.

The hierarchical adjustable algorithm has three intermediate levels, and maintains a hash map in each level, which maps the intermediate result to the final canonical form. This allows the algorithm to finish earlier as soon as the intermediate result hits the hash map. The first level is after deciding the phases of variables (Step 2). The second level is after grouping symmetric variables (Step 4). The simple enumeration (Step 6) executes unconditionally, and its result is used as the key of the third level hash map.

C. *Canonical Example*

Given a function $f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + x_3\overline{x_4})(x_5 \oplus x_6)$, we have the cofactor signature vector $\left(|f|, |f_{x_1}|, |f_{x_2}|, |f_{x_3}|, |f_{x_4}|, |f_{x_5}|, |f_{x_6}|\right) = (14, 4{,}4{,}4{,}10{,}7, 7)$. Since $|f|$<32, the output polarity is positive.

Step 2 perform the phase assignment. Because $|f_{x_1}| = |f_{x_2}| = |f_{x_3}| < \frac{|f|}{2}, |f_{x_4}| > \frac{|f|}{2}, |f_{x_5}| = |f_{x_6}| = \frac{|f|}{2}$, the phase of x_1 , x_2 and x_3 is positive, the phase of x_4 is negative. x_5 and x_6 are balanced. Then, the cofactor signature vector is recalculated as $\left(|f|, |f_{x_1}|, |f_{x_2}|, |f_{x_3}|, |f_{\overline{x_4}}|, |f_{x_5}|, |f_{x_6}|\right) = (14, 4{,}4{,}4{,}4{,}7{,}7)$.

In Step 3, the variables are grouped into two group, $G_1 = (x_1, x_2, x_3, x_4)$, $G_2 = (x_5, x_6)$, where G_2 is balanced.

In Step 4, the variable symmetry is detected, and the input variables are divided into three symmetry classes: $C_1 = [x_1,\ x_2]$, $C_2 = [x_3,\ x_4]$ and $C_3 = \langle x_5,\ x_6 \rangle$. Note that variable x_4 has been negated in Step 2. Then higher-order symmetry is detected, C_1, and C_2 are merged into a second-order symmetry class $C_{12} = [[x_1,\ x_2], [x_3,\ x_4]]$.

After Step 4, the structure of input variables is $G_1 = ([x_1, x_2, x_3, x_4])$, $G_2 = (\langle x_5, x_6 \rangle)$.

Each of the two groups only contains one symmetric class, so no permutation needs to be performed. Because G_2 is balanced, C_3 needs to be flipped. As C_3 is a multiform class, only on variable in C_3 is to be negated.

In Step 7, two transformations are enumerated. The functions after transformation is $f_1 = (x_1x_2 + x_3x_4)(x_5 \oplus x_6)$, $f_2 = (x_1x_2 + x_3x_4)(\overline{x_5} \oplus x_6)$. The truth table $T(f_1)$ =0000 F888 F888 0000H, $T(f_2)$ =F888 0000 0000 F888H. As $T(f_1) < T(f_2)$, the canonical form of f is $\kappa(f) = f_1$.

VI. Experimental Result

Petkovska's hierarchical algorithms [17] are the state-of-the-art classification algorithms, which include the hierarchical algorithm *HierH*, and the exact algorithm *HierE2*. *HierH* is implemented in ABC [23] (command "*testnpn -A 7*"), and the implementation of *HierE2* is not publicly available. Therefore, we use the results from [17], with compensation for the difference of the test environment. The proposed hierarchical adjustable algorithm, referred as *HAdj*, is also implemented in ABC. All experiments ran on a computer with a 3.1GHz Intel Core i5 CPU and 8 GB main memory.

The benchmarks used in the experiments are the same used in [15] and [17]. They are Boolean functions divided into several test suites by their DSD properties [24] (full DSD, partial DSD, and non-DSD), and by the number of input variables.

TABLE I. COMPARISON OF HEURISTIC CLASSIFICATION ALGORITHMS

DSD	# Vars	# Funcs	*HierH* #Classes	*HierH* Time	*HAdj* (Number of classes / Runtime(s) / Exact ratio(%)) Threshold = 0			Threshold = 25			Threshold = 30			Threshold = 35		
Full	6	1M	204	0.06	215	0.07	0	191	0.07	100	191	0.07	100	191	0.07	100
	8	1M	1344	0.15	1393	0.18	0	1274	0.20	99.996	1274	0.22	100	1274	0.21	100
	10	100K	1723	0.05	1729	0.08	0	1713	0.09	99.7	1708	0.09	99.96	1707	0.14	99.999
	12	100K	3157	0.19	3154	0.28	0	3145	0.29	99.6	3140	0.31	99.8	3139	0.42	99.9
	14	10K	891	0.13	898	0.24	0	893	0.24	92.9	887	0.32	95.7	883	0.88	98.8
	16	10K	1057	0.49	1059	0.82	0	1056	0.85	94.6	1055	0.87	95.6	1055	1.05	96.4
Partial	6	1M	2254	0.07	2258	0.09	0	2106	0.09	99.99	2103	0.10	100	2103	0.10	100
	8	1M	14270	0.18	14268	0.28	0	13944	0.36	97.9	13934	0.63	99.5	13923	2.25	99.99
	10	100K	6620	0.07	6593	0.12	0	6520	0.15	96.4	6510	0.22	98.7	6502	1.06	99.6
	12	100K	8545	0.23	8482	0.40	0	8430	0.50	86.9	8419	0.74	94.6	8413	1.87	98.4
	14	10K	2482	0.27	2472	0.49	0	2460	0.55	83.9	2457	0.76	90.3	2454	2.56	94.4
	16	10K	3110	1.19	3101	2.49	0	3089	2.72	77.4	3086	3.04	84.6	3082	5.99	89.0
Non	6	1M	1748	0.04	1744	0.05	0	1674	0.06	99.99	1673	0.06	100	1673	0.06	100
	8	1M	2949	0.09	2936	0.10	0	2857	0.15	98.3	2841	0.28	99.7	2836	0.73	99.9
	10	100K	2016	0.03	2019	0.05	0	1954	0.10	63.8	1924	0.26	75.7	1920	1.11	89.9
	12	100K	1409	0.10	1393	0.13	0	1364	0.17	61.8	1341	0.52	82.4	1327	3.21	91.5
	14	10K	980	0.09	972	0.16	0	968	0.18	42.3	964	0.53	56.6	957	3.64	74.9
	16	10K	282	0.16	281	0.24	0	281	0.26	24.6	281	0.30	41.3	281	0.87	56.9
Geomean				0.13		0.19			0.22			0.32			0.72	

Table I compares the heuristic classification results of *HAdj* with *HierH*. The two columns for *HierH* are the number of classes it produced, and its runtime. For *HAdj*, the results under four different enumeration thresholds are presented. In addition to the number of classes produced, and the runtime, the exact ratio is shown, which is the ratio of the functions, for which exhaustive enumeration was performed, to the total number of functions.

A heuristic classification performs better if it generates fewer classes, that is, closer to the exact result. The pure heuristic version of *HAdj* (Threshold = 0) has similar classification result to *HierH*. It is slightly worse than *HierH* for full DSD functions, and is slightly better than *HierH* for partial DSD and non-DSD functions. *HAdj* is about 1.5x slower than *HierH*. Setting a proper enumeration threshold can improve the classification quality significantly with small runtime penalty. Raising the enumeration threshold increases the exact ratio further, and results in better classification quality, but the runtime grows rapidly

Table II compares the exact classification result of *HAdj* and *HierE2*. *HierE2* is not scalable enough for classifying functions with more than 10 input variables. While *HAdj* can classify full DSD functions with 16 inputs and partial DSD functions with 12 inputs in an affordable runtime. On average, *HAdj* is 31 times faster than *HierE2* for classifying functions with no more than 10 input. *HAdj* still unable to classify the partial DSD test suit with 16 inputs, and the non-DSD test suits with more than 10 inputs. The runtimes of classifying these test suits shown in Table II are estimated using the enumeration cost in Step 5 of the algorithm.

Analyzing the runtime cost of individual functions shows that, for exact classification, a few outlier functions dominate the runtime. As an example, for the partial DSD test suit with 14 inputs, the total classification time is 13.1 hours. Among the 10000 functions in the test suit, 2472 of them are processed for exhaustive enumeration, and the other functions are skipped by the hierarchical mechanism. The top 2 difficult functions cost 9.8 hours, and the next 10 functions cost 2.9 hours. The total runtime of the other 2460 functions is less than 30 minutes.

Comparing the exact runtime to the heuristic runtime (threshold = 35) of *HAdj* also shows the disproportionate runtime of a few functions, as shown in Table III. Take the

TABLE II. CLASSIFICATION RESULTS OF EXACT ALGORITHMS

DSD	# Vars	# Funcs	# Classes	Runtime(s) *HierE2*	Runtime(s) *HAdj*	Speedup
Full	6	1M	191	0.17	0.07	2.38
	8	1M	1274	49.45	0.20	247.75
	10	100K	1707	7691.50	0.63	12208.73
	12	100K	3138	-	1.62	-
	14	10K	882	-	98.82	-
	16	10K	1041	-	550.09	-
Partial	6	1M	2103	0.42	0.10	4.17
	8	1M	13932	113.56	5.03	22.58
	10	100K	6494	5253.51	8.58	612.30
	12	100K	8396	-	923.42	-
	14	10K	2447	-	13h	-
	16	10K	-	-	>6kh*	-
Non	6	1M	1673	0.22	0.08	2.71
	8	1M	2836	16.01	1.83	8.75
	10	100K	1904	3714.13	1272.60	2.92
	12	100K	-	-	>100h*	-
	14	10K	-	-	>100kh*	-
	16	10K	-	-	>500h*	-
Geomean						31.27

*estimated time

TABLE III. RUNTIME COMPARISON OF HEURISTIC AND EXACT CLASSIFICATION

DSD Property	# Vars	Heuristic Time(s)	Exact Time(s)	Runtime Ratio(%)	Function Ratio(%)
Full	10	0.14	0.20	70.0	99.999
	12	0.42	0.63	66.7	99.9
	14	0.88	1.62	54.3	98.8
	16	1.05	98.82	1.1	96.4
Partial	8	2.25	5.03	44.7	99.99
	10	1.06	8.58	12.4	99.6
	12	1.87	923.42	0.2	98.4
	14	2.56	13h	0.005	94.4
Non	8	0.73	1.83	39.9	99.9
	10	1.11	1272.60	0.1	89.9

partial DSD test suit with 12 inputs as an example, the heuristic classification generates 98.4% exact canonical form, and only cost 0.2% of the full exact runtime. In other words, for exact classification, 1.6% of the functions occupy more than 99.8% of the runtime. This is reason for *HAdj* algorithm perform effective heuristic classification. Most of the functions can be processed exactly, only a few difficult functions are processed by the fast heuristic method.

VII. Conclusion

This paper presents an adjustable NPN classification algorithm, which can be either exact or heuristic. As a heuristic algorithm, the proposed algorithm can be adjusted to make a compromise between the runtime and the classification quality. As an exact classification algorithm, the proposed algorithm is faster than state-of-the-art. The main reason of the speedup is that the algorithm takes full advantage of various variable symmetries, especially the proper manipulation of the multiform symmetric groups, which is neglected by many of the existing classification methods.

Exact classification of non-DSD functions with more than 10 inputs is still a difficult problem, which we plan to address in the future.

References

[1] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu,"Efficient FPGA resynthesis using precomputed LUT structures", Proc. FPL 2010, pp. 532–37.

[2] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis", Proc. ICCAD 2012, pp. 597–604.

[3] M. Soeken, L. G. Amarù, P. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing" , Proc. DATE. Dresden, Mar. 2016.

[4] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu, "Generating efficient libraries for use in FPGA resynthesis lgorithms", Proc. IWLS. 2010, pp. 147-154.

[5] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst 25(12), 2894–903 (2006)

[6] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", Proc. ICCAD 2007, pp. 354–61.

[7] A. Mishchenko, R. Brayton, W. Feng, and J. W. Greene, "Technology mapping into general programmable cells", Proc. FPGA 2015, pp. 70–73.

[8] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", Proc. DATE'12, pp. 1579-1584.

[9] W. Feng, J. W. Greene, and A. Mishchenko, "Improving FPGA performance with a S44 LUT structure". Proc. FPGA 2018, pp. 61-66.

[10] U. Hinsberger and R. Kolla, "Boolean matching for large libraries", Proc. DAC1998, pp. 206–211.

[11] D. Chai and A. Kuehlmann, "Building a better Boolean matcher and symmetry detector", Proc. DATE 2016, pp.1079–1084.

[12] S. W. Golomb, "On the classification of Boolean functions", IRE Trans. Circuit Theory, May 1959, vol. CT-6, pp. 176-186..

[13] A. Abdollahi and M. Pedram, "Symmetry detection and Boolean matching utilizing a signature-based canonical form of Boolean functions", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., 2008, 27(6), pp. 1128-1137.

[14] G. Agosta, F. Bruschi, G. Pelosi, and D. Sciuto, "A trasnform-parametric approch to Boolean matching", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., 2009, 28(6), pp. 805-817.

[15] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification", Proc. ICFPT 2013, pp. 310–313.

[16] C.C. Tsai, M. Marek-Sadowska and D. Gatlin. "Boolean function classification via fixed polarity Reed-Muller forms", IEEE Transactions on Computers, 1997 , 46 (2) , pp.173-186.

[17] A. Petkovska, M. Soeken, G. De Micheli, P. Ienne, and A. Mishchenko, "Fast hierarchical NPN classification" , Proc. FPL 2016. pp. 61-64.

[18] C.C. Tsai and M. Marek-Sadowaska. "Generalized Reed-Muller Forms as a Tool to Detect Symmetries", IEEE Transactions on Computers, 1996, 45 (1), pp. 33-40.

[19] N. Kettle and A. King, "An anytime symmetry detection algorithm for ROBDDs", Proc. ASP-DAC 2006, pp. 24-27.

[20] V.N. Kravets and K.A. Sakallah. "Generalized symmetries in Boolean functions", ICCAD 2000, pp. 526-532.

[21] J. Ciric and C. Sechen, "Efficient canonical form for Boolean matching of complex functions in large libraries", IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., 2003, 22(5), pp. 535–544.

[22] Selmer M. Johnson. "Generation of permutations by adjacent transposition", Math. Comp. 17 (1963), 282-285.

[23] Berkeley Logic Synthesis and Verification Group. "ABC: A System for Sequential Synthesis and Verification", [Online]. http://www-cad.eecs. berkeley.edu/~alanmi/abc.

[24] R. Ashenhurst, "The decomposition of switching functions," in Proceedings of the International Symposium on the Theory of Switching, Cambridge, Mass., Apr. 1957, pp. 74–116.