Fast Boolean Matching Based on NPN Classification

Zheng Huang　Lingli Wang
State Key Lab of ASIC and System
Fudan University, Shanghai, China
{11212020020@fudan.edu.cn, llwang@fudan.edu.cn}

Yakov Nasikovskiy
Computer Science Department
CSU Dominguez Hills
lostcorsair@yahoo.com

Alan Mishchenko
Department of EECS
UC Berkeley
alanmi@eecs.berkeley.edu

Abstract—**This paper proposes a fast algorithm for Boolean matching of completely specified Boolean functions. The algorithm is based on the NPN classification and can be applied on-the-fly to millions of small practical functions appearing in industrial designs, leading to runtime and memory reduction in logic synthesis and technology mapping. The algorithm is conceptually simpler, faster, and more scalable than previous work.**

I. Introduction

An NPN class is a set of Boolean functions derived from each other by permuting and complementing the inputs, and complementing the output. Boolean matching determines whether two Boolean functions belong to the same NPN class. In the practical applications, Boolean matching is used to match a function against a library of functions whose implementation or other properties are known or have been pre-computed.

AIG-based logic synthesis and technology mapping implemented in ABC [5] rely on efficient manipulation of small Boolean functions.

For example, in cut-based technology mapping [11], even for relatively small designs and design blocks, millions of structural cuts are repeatedly enumerated. For each cut, a Boolean function is derived and matched against a gate or macrocell library used to implement the function in hardware. In many cases, this library matching procedure is slow because it involves elaborate decomposition checks.

Previous work on Boolean matching differs from the proposed algorithm in that (a) it targets large functions [2], (b) it is inherently BDD-based and therefore not nearly as fast [3][6], (c) it is limited to only permutation of inputs without considering their complementation [10], (d) it is not scalable due to high computational complexity of some computations, such as implicant table generation [8][9].

This paper presents two algorithms: a fast heuristic one and a slower nearly-exact one. They are compared against [8] believed to be state-of-the-art in Boolean matching on a large set of small practical functions derived from industrial designs. The conclusion is that the proposed heuristic algorithm is often faster without degrading quality, while a more elaborate nearly-exact algorithm computes fewer classes than [8]. Moreover, [8] cannot be applied to functions of more than 9 inputs, which limits its practicality. Meanwhile, the proposed algorithm scales to 16 variables and is therefore a better fit for cut-based logic synthesis and technology mapping.

The rest of the paper is organized as follows: Section II contains relevant background. Section III describes the proposed three-step algorithm. Section IV shows experimental results. Section V concludes the paper.

II. Background

A. Boolean function

This paper deals with completely specified Boolean functions, $f(X)$: $B^n \to B$, $B = \{0,1\}$. The support of F, denoted $supp(F)$, is the set of variables whose value influences the output value of F. Boolean function is often represented by its *truth table*, which is a string of 2^n bits. The i-th bit of the truth table is equal to the output value of the function when its inputs take values equal to the binary digits of integer number i.

Definition 1: A *cofactor* of function $F(\ldots, x_i, \ldots, x_j, \ldots)$ with respect to (w.r.t) variables x_i and x_j, is the function derived by substituting into F of specific values for x_i and x_j. For example, cofactor of F w.r.t. $x_i = 0$ and $x_j = 1$ is function $F(\ldots, 0, \ldots, 1, \ldots)$ denoted F_{01}.

Definition 2: Two variables, x_i and x_j, of F are *symmetric* if the function does not change when x_i and x_j are swapped

$$F(\ldots, x_i, \ldots, x_j, \ldots) = F(\ldots, x_j, \ldots, x_i, \ldots).$$

The definition of symmetry translates into the following requirements for the cofactors: $F_{01} = F_{10}$. When the symmetric variables are collected in one group, the group is defined as a symmetric group.

Theorem 1: Assume that variable a is symmetric with variable b, and variable b is symmetric with variable c, then variable a must be symmetric with variable c, which is defined as the *transitivity* of symmetric variables.

Definition 3: In some cases, each variable in a symmetric group is not symmetric with each variable in another symmetric group, but the two symmetric groups are symmetric, that is, can be swapped without changing the function. Such relationship between two or more groups of symmetric variables is called a higher-order symmetry.

For instance, consider function $F = ab \oplus cd$. Variable pairs (a, b) and (c, d) form symmetric groups. Nevertheless, a is not symmetric with c or d. Likewise for b. However, (a, b) is symmetric with (c, d), which creates a higher-order symmetry denoted as $((a, b), (c, d))$.

B. NPN canonical form

Changing the order of two variables in the support of a Boolean function is called a *swap* ($ab \to ba$). Replacing a variable by its complement is called a *flip* ($a \to\ !a$). There are eight transformations of the function performed by swapping and flipping pairs of adjacent support variables: ab, $a!b$, $!ab$, $!a!b$, ba, $b!a$, $!ba$, $!b!a$.

This observation can be generalized as follows: For an N-variable Boolean function, there are 2^N ways of transforming it by flipping its variables and $N!$ ways of transforming it by swapping its variables. Additionally, there are two polarities of the function derived by complementing its output. In total, there are $M = 2^{N+1} * N!$ transformations of the function derived by swapping its inputs and flipping its inputs and output.

Definition 4: Consider the set of all Boolean functions derived by M transformations of a Boolean function F, as described above. These functions constitute the *NPN class* of function F. The *NPN canonical form* of function F is one function belonging to its NPN class, also called the *representative* of this class.

The representative is selected uniquely among the functions belonging to the given class. For example, in some applications, the representative is selected as the function whose truth table, considered as an integer number, has the smallest numeric value. In other applications, the representative is uniquely selected by some deterministic algorithm.

The number of NPN classes is much smaller than the number of Boolean functions. For example, all 2^{16} Boolean functions of 4 variables split into 222 NPN classes.

C. Boolean network

Definition 5: A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) whose nodes correspond to logic gates and edges to wires connecting the nodes. It is assumed that each node has a unique integer called *node ID*.

Definition 6: Two Boolean functions are *NPN-equivalent* if they have the same NPN form, that is, one of them can be obtained from the other by swapping and flipping the inputs and the output. An example of NPN-equivalent functions is given in Figure 1.

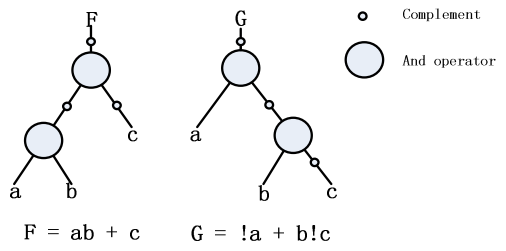

Fig. 1. NPN-equivalent functions F and G.

As shown in Figure 1, *F* = *ab* + *c*, and *G* = !*a* + *b*!*c*, are NPN-equivalent because *F* can be transformed into *G*, when variable *a* is swapped with *c,* and variable *b* is swapped with *c*, while variables *a* and *c* are complemented. As a result, function *F* is isomorphic to *G* after a flipping/swapping of inputs/outputs. Therefore, functions *F* and *G* are NPN-equivalent.

Theorem 2: Two NPN-equivalent functions F and G have the same symmetric group structure including the number of groups and the size of each group. The same is true about higher-order symmetries.

Proof: If function *F* is NPN-equivalent to function *G*, these two functions belong to the same NPN class. Moreover, *F* and *G* can be transformed to the representative of this NPN class, which is denoted as F^{r}. So it can be deduced that *F* must have the same symmetric group with F^{r}, and likewise for *G*. Therefore, *F* must have the same symmetric group structure as *G*. The same is true for higher-order symmetries. **Q.E.D**.

D. Disjoint-support decomposition

Disjoint-support decomposition (DSD) [1] represents a completely-specified Boolean function as a tree of nodes with non-overlapping supports and inverters as optional attributes on the edges. This tree is called the *DSD structure* of the function. DSD is *full* if the DSD structure is composed of elementary gates (AND, XOR, MUX). If a given Boolean function has a full DSD, the DSD structure is unique up to the permutation of inputs and positions of inverters. For example, DSD of function *F* = *ab* + *cd* is full and its structure contains three ANDs and three inverters: *F* = !(!(*ab*) ∧ !(*cd*)).

DSD plays an important role because (a) it is a canonical representation, (b) efficient algorithms for computing DSD from the BDD or the truth table are known [4], and (c) the majority of Boolean functions of 4-16 inputs appearing as cut-functions in hardware designs have full or partial DSD.

In this paper, we use the existence of a full or partial DSD as a criterion to partition Boolean functions into several classes used to test the proposed algorithm.

III. Algorithm

This section describes the three-step algorithm for computing exact NPN-canonical form for Boolean functions of 6-16 inputs represented as a truth table. Only the first step was used in previous work [2][8][13], but symmetric variables were not detected, which degraded the results produced by the algorithm.

A. Ordering variables using the count of 1s

First, the number of 1s (the number of onset minterms) in the truth table is computed. The polarity of the function is determined so that the truth table contained fewer 1s than its complement. If the number of 1s is tied, both polarities are considered and a unified canonical form is derived.

Second, polarity of each variable is determined by considering the number of 1s in the cofactors of the function w.r.t. this variable. The negative cofactor is required to contain fewer 1s than the positive cofactor. In the case of a tie, polarity is not changed.

Third, the ordering of variables is determined using the number of 1s in the cofactors w.r.t. each variable. For this, we require that variables were ordered in the increasing order of the number of 1s in the negative cofactor. If there is a tie, polarity is not changed.

Theorem 2 from Section II C indicates that detecting and grouping the symmetric variables can help uniquely identify the NPN-equivalent class. A tie in the number of 1s for a group of variables, which are called a *tied group*, indicates that some or all of them may be symmetric. In the forthcoming computations, each symmetric group is moved as a single entity. For example, when a variable from a group is swapped with another variable, the whole symmetric group is moved to the place where the given variable should be placed. In the end, the resulting truth table is a semi-canonical form of the function. This form is not the exact NPN canonical form because non-symmetric variables whose polarities and/or positions in the order are tied may not be uniquely determined.

If polarity and order of each variable is uniquely determined, the algorithm terminates. Otherwise, the next two steps are used to gradually uniqify the tied variables.

B. Ordering variables using pairwise permutations

Ordering variables by pairwise variable permutations is conceptually similar to dynamic variable reordering in a BDD using adjacent variable swaps. There are two differences: (1) the cost is not the BDD node count but the integer value of the truth table (for example, the smaller value is preferable), and (2) when variables are reordered by the proposed method, the eight different configurations are considered (*ab*, *a*!*b*, !*ab*, !*a*!*b*, *ba*, *b*!*a*, !*ba*, !*b*!*a*) instead of only two (*ab* and *ba*). The configuration leading to the smallest integer value of the truth table is chosen and the truth table is updated accordingly.

Given symmetric variable groups computed in Section III A, each symmetric group is treated as one entity in the following procedures. Therefore, reordering of symmetric groups is performed instead of reordering of variables.

If polarity and order of each symmetric group of all the tied groups is uniquely determined in this step, the algorithm terminates. Otherwise, the final step is performed to unify polarity and order of the tied variables.

C. Exhaustive reordering of tied variables

This step is applied to the variables (or groups of symmetric variables) in the tied groups. Groups of tied variables are considered in some order. Each group is subjected to exhaustive swaps and flips to determine the configuration that leads to the smallest integer value of the truth table. However, the runtime of performing exhaustive swaps and flips on all the variables in tied groups is very high. Assuming that the number of variables in one tied group is N, then the time complexity of exhaustive swaps and flips is proportional to $N!*2^N$. Decreasing N can substantially reduce the runtime. Hence the exhaustive procedure can be optimized as follows.

a) Detecting higher-order symmetric groups.

Symmetric groups are computed as shown in Section III B. As a result, higher-order symmetric groups can be detected and collected among these symmetric groups. Similar to collecting the symmetric groups in Section III A, higher-order symmetric groups can be collected via swapping two symmetric groups in one tied group. If the truth table does not change when two symmetric groups are swapped, these two symmetric groups belong to one higher-order group.

b) Rebuilding the tied groups according to the size of each higher-order symmetric group.

In this step, the tied groups detected in Section III A are divided into smaller tied groups containing the same-size higher-order symmetric group. The reason why the tied groups need to be divided is that, as depicted above, the time complexity is $N!*2^N$, so when group of size N is divided into smaller groups of size N1 and N2, the time complexity is reduced to $(N1!*2^{N1})*(N2!*2^{N2})$. In order to rebuild the tied groups, we first perform sorting of the higher-order symmetric groups collected in the last step *a)* in ascending of the size of higher-order symmetric group in one tied group. After dividing the same-size higher-order symmetric group into one tied group, the new tied groups are formed.

c) Performing exhaustive swaps and flips of higher-order groups marked in step b).

In this step, we use higher-order symmetric groups tagged in last step *b)*. For all the higher-order group exhaustive swaps and flips are performed.

When all the groups have been processed, the final value of the truth table is returned as the resulting canonical form. For most practical functions, the runtime can be decreased effectively after the above optimizations. The result after the above optimizations is exact NPN canonical form.

D. Heuristic version of the algorithm

The above algorithm is exact and potentially slow due to the exhaustive enumeration of swaps and flips for groups of tied variables in the final step, described in Section III C.

The algorithm can be made heuristic by skipping the final step. The resulting form is semi-canonical, rather than exactly canonical. It means that, for some functions of the same NPN class, the algorithm produces different semi-canonical forms.

A semi-canonical form is similar to the exact canonical form in that it can be used to reduce the size of the cache, as described in the introduction to this paper. Using semi-canonical form increases the cache size but the result is still correct. Experiments show that the increase in the cache size due to skipping the computationally intensive final step, does not exceed 3%.

IV. Experimental results

The proposed algorithm is implemented in ABC [5] and integrated into command *testnpn* designed to compare runtime and quality of different canonical forms. Command *testnpn –A 5 <inputfile>* implements the heuristic version of the algorithm, while the command *testnpn –A 6 <inputfile>* implements the exhaustive version.

The runtime is measured as the time needed to canonicalize the given number (say, one million) of N-input functions. The quality is measured as the number of unique classes computed by the algorithm. The closer is this number to the exact number of NPN classes for the given set of functions, the better is the algorithm.

The correctness of the algorithm is verified by making the procedure return the canonical configuration (input permutation and complementation as well as output complementation). The inverse of this configuration is applied to the canonical form and the result is checked to be identical to the original function.

A. Profiling the algorithm

Tables 1, 2, and 3 contain experimental results for practical functions of different number of inputs. To reduce the amount of data, only functions with even support sizes are considered in these tables, while in general the algorithm works for any function up to 16 variables.

The functions are partitioned into three sets using disjoint-support decomposition [1][4]: fully decomposable, partially decomposable, and non-decomposable. This is done because performance of the algorithm on each set is different.

In particular, many fully decomposable functions contain groups of symmetric variables. For example, $F = ab + cd$ is fully decomposable and has symmetric variables pairs (a, b) and (c, d). The presence of symmetry is exploited by the proposed algorithm to compute the exact NPN class. Partially decomposable and non-decomposable functions have fewer symmetries. Consequently, the heuristic algorithm makes a tradeoff between runtime and quality for these functions. Further, as stated in Section III C, the time complexity of exhaustive reordering is $N!*2^N$, so fewer symmetries in partially decomposable and non-decomposable functions result in large N and the runtime increases accordingly. Therefore, reordering is not exhaustive when these two kinds of functions are processed. As a result, the so-called exhaustive version of our algorithm does not compute exact NPN classes for partially decomposable and non-decomposable functions.

The columns “Step 1”, “Step 2”, and “Step 3” (and “T1”, “T2”, and “T3”) list the number of classes (and the runtime) of the three steps of the proposed algorithm, described in Sections III A, III B and III C, respectively. Two lines at the bottom of each table contain arithmetic and geometric averages of the entries listed in the corresponding columns.

As an example, consider applying the algorithm to 100K 10-input fully decomposable functions in Table 1. Comparing columns “Step 1” and “Step 2” shows the reduction of the NPN classes from 2580 to 1723. This is a 33% reduction while the runtime increases from 0.43 to 0.44 sec is very small. Comparing columns “Step 2” and “Step 3” shows the reduction in the number of NPN classes from 1723 to 1707 with the runtime increase from 0.44 to 1.23 sec. The number of NPN classes is reduced by 1% while the runtime is increased more than 2 times. The resulting number of NPN classes is exact.

B. Comparing with previous work

Columns “[8]” and “T” in the tables show the number of classes and the runtime produced by Boolean matcher [8]. Finally, the last column shows the number of true NPN classes produced by the exact and slow computation based on the complete enumeration of all configurations. Since the matcher in [8] does not scale to functions of more than 9 inputs, some entries in the tables contain dashes.

The results show that both the proposed algorithm and [8] are exact for fully decomposable functions and heuristic for other functions. The proposed algorithm is on average faster than [8] and computes fewer classes. Another difference is that the proposed algorithm scales to functions up to 16 inputs, which is important for practical applications, such as mapping into LUT structures [12] and technology-independent synthesis [13].

The runtime of the exact NPN algorithm was two or three orders of magnitude higher than the runtime of the heuristic algorithm. This confirms the intuition that exact NPN matching remains a hard computational problem for Boolean functions without full DSD.

V. Conclusions and future work

This paper presents a Boolean matching algorithm, which efficiently computes a semi-canonical form for completely specified Boolean functions of 6-16 inputs arising in industrial designs. The algorithm is also applicable to functions of 18-22 inputs, but the runtime degrades because of the use of truth tables that grow exponentially in the number of variables.

An early version of the proposed algorithm was used in [13] to reduce the cache size. However, when the algorithm was improved, as discussed in this paper, the size of a library of pre-computed logic structures for two million 6-variable functions was reduced about 3 times.

The following directions will be explored as future work:

- Investigate the relationship between disjoint-support decomposition and canonical form computation. In particular, given a full or partial DSD, it is possible to by-pass some steps in the canonical form computation.
- Use the algorithm to enhance mapping into both standard cells [6] and LUT structures [12]. Indeed, matches for each function can be precomputed and saved for each semi-canonical class, resulting in reduced runtime.

VI. Acknowledgement

This paper is supported by National Natural Science Foundation of China (61131001, 61171011).

REFERENCES

[1] R. L. Ashenhurst, “The decomposition of switching functions”. *Computation Lab*, Harvard University, 1959, Vol. 29, pp. 74-116.

[2] A. Abdollahi and M. Pedram. “A new canonical form for fast Boolean matching in logic synthesis and verification”. *Proc. DAC’05*, pp. 379–384.

[3] L. Benini and G. De Micheli. “A survey of Boolean matching techniques for library binding”. *ACM TODAES*, 1997, Vol. 2(3), pp. 193-226.

[4] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD ‘97*, pp. 78-82.

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

[6] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD'05*, pp. 519-526.

[7] J. R. Burch and D. E. Long. “Efficient Boolean function matching”. Proc.’92, pp. 408–411.

[8] D. Chai and A. Kuehlmann, “Building a better Boolean matcher and symmetry detector”, *Proc. DATE’06*.

[9] J. Ciric and C. Sechen. “Efficient canonical form for Boolean matching of complex functions in large libraries”, *IEEE Trans. CAD*, May 2003, Vol. 22(5), pp. 535–544.

[10] D. Debnath and T. Sasao. “Efficient computation of canonical form for Boolean matching in large libraries”. *Proc. ASP-DAC’04*, pp. 591-596.

[11] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

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

[13] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis", *Proc. ICCAD'12*, pp. 597-604.

Table 1: Quality/runtime tradeoff for practical N-input fully decomposable functions.

N	Functions	Step 1	T1, sec	Step 2	T2, sec	Step 3	T3, sec	[8]	T, sec	Exact NPN
6	1M	1184	0.52	204	0.54	191	2.97	191	5.18	191
8	1M	4109	1.32	1344	1.74	1274	5.59	1274	21.10	1274
10	100K	2580	0.43	1723	0.44	1707	1.23	-	-	1707
12	100K	4205	2.41	3157	2.79	3138	3.58	-	-	3138
14	10K	1080	0.93	891	0.96	882	2.03	-	-	882
16	10K	1293	4.28	1057	4.30	1041	8.03	-	-	1041
Ave		1.000	1.000	0.593	1.096	0.584	3.059			0.584
Geo		1.000	1.000	0.517	1.090	0.503	2.738			0.503

Table 2: Quality/runtime tradeoff for practical N-input partially decomposable functions.

N	Functions	Step 1	T1, sec	Step 2	T2, sec	Step 3	T3, sec	[8]	T, sec	Exact NPN
6	1M	5142	0.58	2254	0.73	2145	6.23	2138	10.24	2103
8	1M	22696	1.52	14270	3.15	14000	15.15	13983	51.89	13932
10	100K	8893	0.59	6620	0.74	6505	2.68	-	-	6494
12	100K	10552	2.67	8545	3.45	8430	10.39	-	-	8412
14	10K	3508	1.08	2482	1.37	2458	3.33	-	-	2433
16	10K	3590	4.86	3110	7.04	3085	17.95	-	-	3061
Ave		1.000	1.000	0.699	1.432	0.687	5.986			0.683
Geo		1.000	1.000	0.683	1.407	0.670	5.275			0.665

Table 3: Quality/runtime tradeoff for practical N-input non-decomposable functions.

N	Functions	Step 1	T1, sec	Step 2	T2, sec	Step 3	T3, sec	[8]	T, sec	Exact NPN
6	1M	2526	0.63	1748	0.65	1692	6.48	1742	8.10	1673
8	1M	3879	1.57	2949	2.25	2887	12.49	2899	73.30	2836
10	100K	2884	0.55	2016	1.25	1974	4.89	-	-	1905
12	100K	1833	2.53	1409	3.80	1381	13.56	-	-	1368
14	10K	1301	1.14	980	1.43	968	4.20	-	-	957
16	10K	312	5.17	282	8.23	282	24.62	-	-	273
Ave		1.000	1.000	0.763	1.514	0.750	6.823			0.735
Geo		1.000	1.000	0.760	1.470	0.746	6.395			0.732