Efficient Canonical Form for Boolean Matching of Complex Functions in Large Libraries

Jovanka Ciric
Synplicity Inc.
935 Stewart Drive
Sunnyvale, CA 94085
jovanka@synplicity.com

Carl Sechen
University of Washington
Dept. of Electrical Engineering, Box 352500
Seattle, WA 98195
sechen@ee.washington.edu

Abstract

A new algorithm is developed which transforms the truth table or implicant table of a Boolean function into a canonical form under any permutation of inputs. The algorithm is used for Boolean matching for large libraries that contain cells with large numbers of inputs and implicants. The minimum cost canonical form is used as a unique identifier for searching for the cell in the library. The search time is nearly constant if a hash table is used for storing the cells' canonical representations in the library. Experimental results on more than 100,000 gates confirm the validity and feasible run-time of the algorithm.

1. Introduction

An important part of logic synthesis and verification is testing if two Boolean functions are equivalent. During technology-mapping, Boolean matching is used to check if part of a network implements a gate in the library. A Boolean function is usually represented in the form of a binary decision diagram (BDD) or a truth table.

Boolean matching is also significant in the library-free technology-mapping problem. The "fluid" library of cells typically has a constraint on the number of parallel and serial devices. The size of the fluid library is very large. Therefore, a cell generator is needed to lay out the cells. Boolean matching is used to check if a cluster of a network that satisfies the parallel and series constraint (and therefore forms a valid gate in the library) has already been laid out by the cell generator.

In the most general case, Boolean matching implies resolving if two Boolean functions are the same under negation of inputs, permutation of inputs or negation of outputs.

Boolean functions that are equivalent under negation of inputs form an *N*-equivalent class, under permutation of inputs a *P*-equivalent class, and under any of the three stated conditions an *NPN*-equivalent class.

Previous work in Boolean matching was done using signatures and canonical representations on BDDs or truth tables [1, 2, 3, 4, 6, 7]. A signature is a description of an input variable that is independent of the permutation of the inputs of a Boolean function [1]. A necessary condition for equivalency of two Boolean functions is that their signatures are the same. However, that is not a sufficient condition, because one signature can describe two or more different functions. It was observed in [1] that regardless of the quality of the signatures used on ROBDDs, they always failed to uniquely identify certain variables. These variables have symmetry properties and they form an aliasing group.

If the size of the aliasing group is k, then $k!$ correspondences need to be additionally tested. For a set of benchmarks, the number of permutations can be more than 100,000 [1]. Burch and Long developed a canonical form for phase matching (equivalency under negation of inputs), but for input permutation only a semi-canonical form was proposed [3]. The semi-canonical forms can be different for some permutation equivalent functions.

In previous work [4, 6, 7], it was pointed out that the algorithms were targeted for logic functions with small numbers of inputs (less than 10). Our algorithm handles Boolean functions with large numbers of inputs and product terms (easily handling 25 inputs and more than 1000 product terms).

We developed an algorithm that guarantees a unique canonical form of the truth table or implicant table, under any permutation of inputs and for any type of symmetric inputs. Moreover, our algorithm can be used for obtaining a unique ordering of variables in ROBDDs. After a unique variable ordering is established and ROBDDs are constructed, two ROBDDs are compared in nearly constant time if a hash table is used for storing the ROBDDs. A previous attempt at finding unique variable orderings for ROBDDs was done using signatures and it failed for some cells in the library [5]. Finally, our algorithm for computing a canonical form under input permutation is more efficient than [4] for functions with large

number of inputs, because the canonical representation in [4] has a length of 2^n for an n-input function.

2. Problem Formulation

In the technology-mapping step, we used the commands in SIS [9] for manipulating the network. The Boolean functions were represented with the sum-of-products form, as in the *blif* [9] format. We assume that the library consists of positive unate, negative unate and binate functions. The binate functions, such as XORs and MUXs, are represented with truth tables containing only the rows where the function's output is 1. The positive and negative unate functions are represented with an implicant table. An implicant table consists of rows of implicants where the output is 1. For unate functions, the sum-of-products (SOP) form consisting of all prime implicants is minimal and unique [10]. Therefore, the implicant table of a unate function will be unique. Because the minimized SOP form is more compact than a truth table, we used it as the unate function's representation for Boolean matching. Examples of a truth table and implicant tables for positive and negative unate functions are shown in Figure 1.

Usually, a vast majority of the functions in a library are unate. Therefore the negation of inputs and output in the Boolean matching problem is unnecessary for these functions. In this case, our cluster function (a portion of the unmapped network that is collapsed into a single node) is made unate prior to calculating its canonical form, by adding inverters at its inputs, where necessary. If there are binate cells in the library and the cluster function does not match any of the unate cells, the cluster function is transformed into a truth table and checked for equivalency.

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

(a) $f = a\bar{c} + bc$

a	b	c	d
0	-	0	-
0	-	-	0
-	0	0	-
-	0	-	0

(b) $g = \overline{ab + cd}$

a	b	c	d
1	1	-	-
-	1	1	-
-	-	1	1
1	-	-	1

(c) $h = ab + bc + cd + ad$

Figure 1. Examples of (a) a truth table, (b) a negative unate function implicant table, (c) a positive unate function implicant table

1	. .	j	. .	N	
2^{M*N-1}	. .	$2^{(M-1)*N+N-j}$	. .	$2^{(M-1)*N}$	1
. .	. .	. .	. .	. .	:
$2^{(M-i)*N+N-1}$	. .	$2^{(M-i)*N+N-j}$	. .	$2^{(M-i)*N}$	i
. .	. .	. .	. .	. .	:
2^{2N-1}	. .	2^{2N-j}	. .	2^{N}	M-1
2^{N-1}	. .	2^{j}	. .	2^{0}	M

Figure 2. Weight assignment for a truth table

For a given truth or implicant table, we assign a weight for each cell in the table. Weight assignment for a table with M rows and N columns is presented in Figure 2.

The columns correspond to the inputs of the function and the rows correspond to the minterms, or implicants in the case of the SOP form. Each cell in the truth table has a zero or one, depending on whether the input is complemented or not complemented in the minterm. For the SOP form, don't cares are assigned a value of zero. If a variable appears in the implicant, it is assigned a value of one in the table. Negative unate functions contain don't cares and zeros in the implicant table. In that case, don't cares are set to zero and zeros are transformed to ones. We would like to find a unique canonical representation of the truth table under arbitrary permutations of its rows and columns. The cost function for any permutation of rows and columns can be computed using the following formula (1):

$$\text{cost}(\mathbf{pr}, \mathbf{pc}) = \sum_{i=1}^{M} \sum_{j=1}^{N} v(pr[i], pc[j]) \cdot 2^{(M-pr[i])\cdot N+N-pc[j]}$$

Permutation vectors ***pr*** and ***pc*****,** for rows and columns respectively, are defined as ***pr*** = (π(1), π(2),.., π(M)); ***pc*** = (π(1), π(2),.., π(N)); π: {1, 2, .., n}$\rightarrow${1, 2, .., n} is the permutation function, and v (i,j) $\in$ {0, 1}.

The cost function represents a decimal equivalent of a binary number obtained by concatenating the rows of the table next to each other, starting from the top of the table to the bottom. For example, different costs for the *aoi*22 function under permutation of inputs are depicted in Figure 3.

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

1010100101100101
= 43365

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

0011010110101100
= 13740

Figure 3. Different costs according to formula (1) for a function *aoi*22 under permutation of inputs

Theorem 1. There exists a canonical form of a truth table, which has a minimum cost (according to formula 1), among all possible permutations of rows and columns.

Proof. The cost function in formula 1 represents a decimal equivalent of a binary number, obtained by concatenating the rows together starting from the top row to the bottom row. Permutations of rows and columns create a discreet, finite set of integers, which has a minimum. A table with the minimum cost is the canonical form.

3. Computing the Minimum Cost Canonical Form

The cells in the upper left corner of the table have the highest weight and the cells in the lower right corner have the lowest weight. The weight decreases going from left to right, and top to bottom. To minimize the cost function, one would like to swap the rows and columns of the table, such that the ones in the table are placed to the right and to the bottom. Also,

since the succeeding smaller weights for a cell are along the same row to the right, we minimize the cost of the table starting from the top rows to the bottom.

We use a branch-and-bound algorithm, with a very tight bounding function, for obtaining the canonical form. First, we find a row with the minimum number of ones and place it on the top. The columns are swapped such that all zero elements in that row are on the left, and all ones on the right. After rearranging the columns, the top row is fixed, and we proceed to find the next minimum cost row. However, the previously fixed rows determine which columns can be swapped in the later steps. The block of columns with the same elements in the fixed row can be swapped without changing the cost of the row. But swapping a column that contains a zero with a column that contains a one in the fixed row will change the cost of the row. Therefore, we insert a column boundary at the transition from zeros to ones in the minimized row. Only columns inside the column boundaries can be swapped.

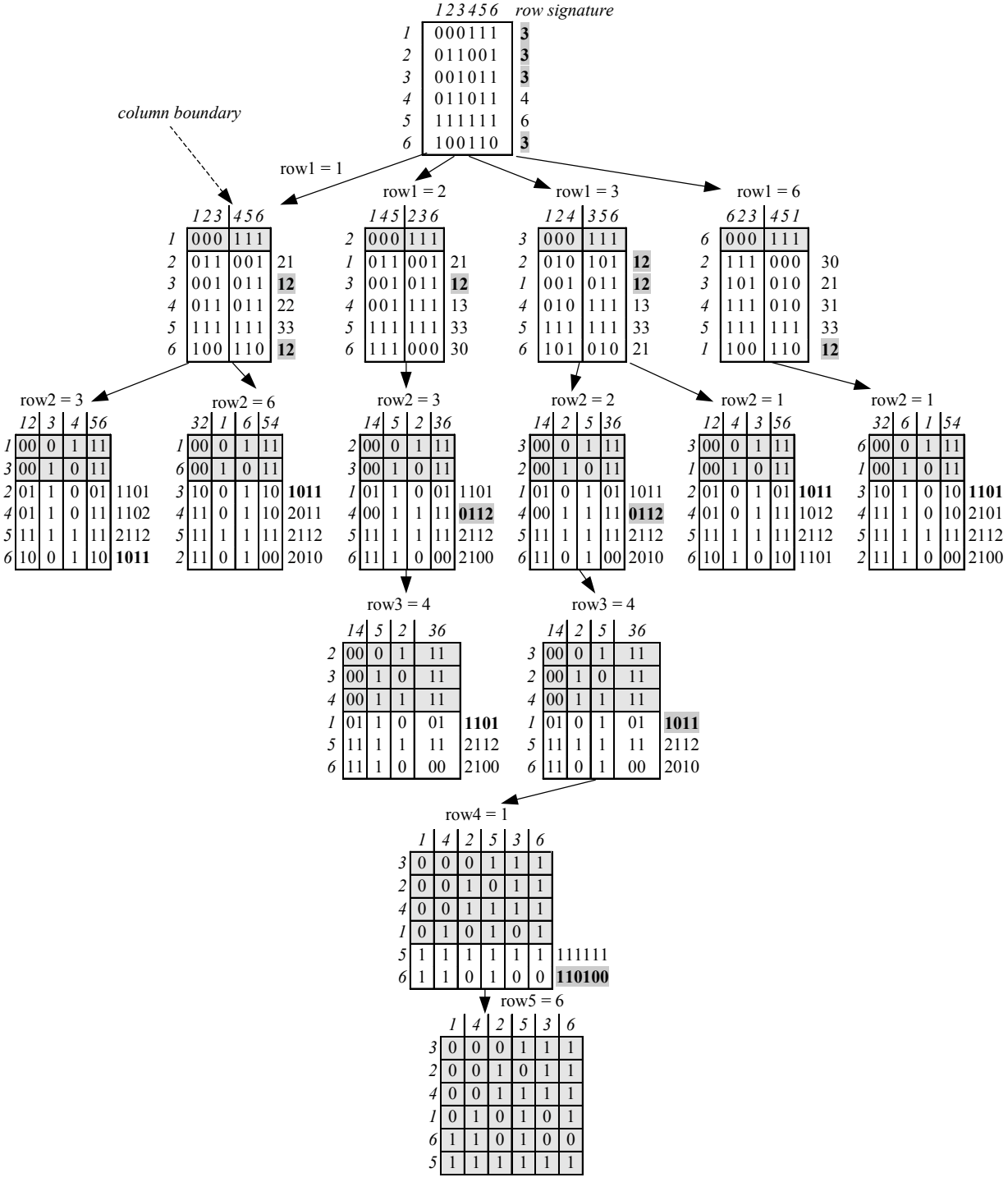

Figure 4. Example of transforming a truth table into its minimum cost canonical form

MCCF (M, N, $\mathbf{A}$, $\mathbf{A}_{canonic}$)

Input: Positive integers M, N (M = number of rows, N = number of columns), truth table $\mathbf{A} \subseteq \{0, 1\}^{M \times N}$

Output: $\mathbf{A}_{canonic}$ canonical form of table $\mathbf{A}$.

$cr = 1$; /* current_row */

$LL \leftarrow \mathbf{A}^1 = \mathbf{A}$; /* linked list of tables */

$K = 1$; /* K = number of tables $\mathbf{A}^i$, $i = 1,.., K$ in the linked list */

$h = 1$; /* h = number of column boundaries */

/* column boundaries $c_1,.., c_h$, satisfy $h \le N$ and $c_1 = 1 < c_2 <..< c_h \le N$ */

1. $min\_signature = \infty$. For each $i = 1,.., K$ do steps 1.1. through 1.3.
   1.1. Let $c_{h+1} = N + 1$
   $s_{kj}^{\ i}$ = number of 1s in columns c_j, $c_j + 1,.., c_{j+1} - 1$ of $\mathbf{A}^i$, $k = cr,.., M$, $j = 1,.., h$
   $S_k^{\ i} = (s_{k1}^{\ i}, s_{k2}^{\ i},.., s_{kh}^{\ i})$, $k = cr,.., M$, $i = 1,.., K$
   1.2. Find minimum $S_{min}^{\ i}$ from all $S_k^{\ i}$.
   Let $min\_row\_set\ (\mathbf{A}^i) = \{r \subseteq \{cr,.., M\} : S_r^{\ i} = S_{min}^{\ i}\}$
   1.3. If ($S_{min}^{\ i} < min\_signature$) then $min\_signature = S_{min}^{\ i}$.
2. $K' = K$; For each $i = 1,.., K$ do steps 2.1. and 2.2.
   2.1. If ($S_{min}^{\ i} = min\_signature$) then for each $r \subseteq min\_row\_set\ (\mathbf{A}^i)$, do steps 2.1.1. through 2.1.3.
      2.1.1. Permute the columns $c_j,.., c_{j+1} - 1$ ($j = 1,.., h$) of $\mathbf{A}^i$ so that 0s are followed by 1s between the column boundaries of row r. Swap minimum row r with current row cr. Let $\mathbf{A}^i{}_r$ denote the resulting table.
      2.1.2. Insert new column boundaries between the existing boundaries c_j, at the columns of transition between zeros and ones, in row r. Let h denote the new number of column boundaries.
      2.1.3. Add $\mathbf{A}^i{}_r$ to the linked list LL; $K' = K' + 1$.
   2.2. Remove $\mathbf{A}^i$ from the linked list LL; $K' = K' - 1$.
3. $K = K'$; $cr = cr + 1$; If ($cr \le M$) go to step 1.

$\mathbf{A}_{canonic} = \mathbf{A}^i$, $\mathbf{A}^i \in LL$, $\forall i$.

Definition 1. Column boundaries partition a row into sets (blocks) of columns such that column permutations may only take place within the sets.

Definition 2. A row signature is a sequence of numbers where each number represents the number of ones in the corresponding block of columns, where the blocks are examined from left to right for the row.

The algorithm for transforming a table into its minimum cost canonical form (*MCCF*) will be explained with an example. Figure 4 shows the truth table of a Boolean function with 6 inputs and 6 minterms

$$f = x_1\bar{x}_2\bar{x}_3x_4x_5\bar{x}_6 + (\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3)x_4x_5x_6 + (x_2 + x_5)\bar{x}_1x_3\bar{x}_4x_6 .$$

At the beginning, we look at the whole table. The row signatures represent the number of ones in each row. The minimal row signatures are written in bold font. Since we have multiple rows with a minimum row signature, they are all candidates for the first row in the canonical form. This is the branching point of the algorithm. Four tables are created, each with a different minimum row as its first row. The columns of the tables are permuted such that all ones in the minimized row are grouped at right of the table, and all zeros at the left. A column boundary is placed at the transition from zeros to ones in the minimized row. The first row is then fixed. The tables are stored in a linked list.

For each table in the list, row signatures are calculated based on the number of ones inside the column boundaries. Row signatures are written on the right side of the table. The minimum row signature is noted for each table. The smallest minimum row signature among all tables in the list determines the bounding function in the algorithm. The smallest signature in the list of tables is shaded in Figure 4. Only the tables with a minimum row signature equal to the minimum among all the tables in the list are kept.

At the end of the algorithm, the list contains the canonical forms of the initial truth table. In this example, there is a unique solution for permuting the rows and columns to obtain the canonical form. In the general case, different permutations of rows and columns can lead to the same minimum cost canonical form for the truth table. *MCCF* is the formal definition of the minimum cost canonical form algorithm. The following lemmas and a theorem are proved in the appendix.

Lemma 1. The column boundaries for all tables A^i in the list LL are identical at the end of each iteration of the algorithm (step 3).

Lemma 2. Subtables made of the top minimum signature rows of tables A^i in the list LL are identical.

Theorem 2. The algorithm *MCCF* produces a canonical form of a truth table A with a minimum cost according to formula 1.

The run time of the algorithm depends largely on the number of rows with a minimum signature, and consequently on the number of tables in the list *LL*. Figure 5 shows a possible execution of the algorithm. The algorithm branches whenever there are multiple rows with the same minimum row signature. Some branches may finish if the minimum row signature for the table on that branch is greater than the minimum row signature of all the tables in the list. However, because of symmetric inputs, there could be branches that are never trimmed by the

bounding function. Equivalent branches worsen the run time of the algorithm, so they should be detected and eliminated.

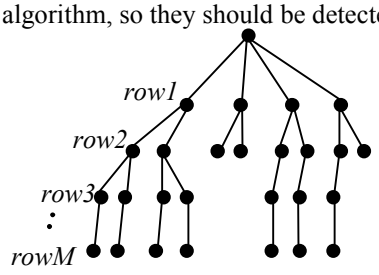

Figure 5. Execution of *MCCF* algorithm

The use of symmetries to avoid equivalent branches is described in the next section.

4. Use of Symmetries

Let's look at an example of a Boolean function that exhibits input variable symmetry. Figure 6 shows the execution of the algorithm *MCCF* for the given truth table. Numbers at the nodes of the execution tree represent the rows with minimum signature from top to bottom.

$$y = (a + b) \cdot (c + d)$$

	a	*b*	*c*	*d*
1	1	0	1	0
2	0	1	1	0
3	1	0	0	1
4	0	1	0	1

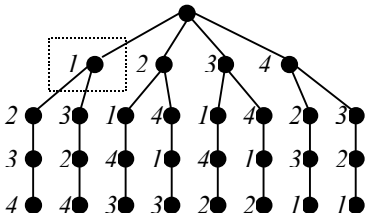

Figure 6. An example of the execution tree for a function with variable symmetry

Figure 7 presents one stage in the algorithm for the example in Figure 6 (the node contained in the rectangle). After fixing the first row, there are two rows (2 and 3) with the same minimum row signature. Row 3 can be obtained from row 2 by swapping the columns *d* and *b,* and *c* and *a.* This swapping is inside the existing column boundaries. Furthermore, the rest of the table is the same regardless of which row (2 or 3) is selected first. Therefore, branching of the algorithm on both of the rows 2 and 3 is unnecessary, because the tables on these branches are identical and it will lead to the same minimal canonical forms. We will call rows 2 and 3 symmetric rows.

Definition 3. Two rows in a truth table are symmetric if a) they have the same row signature and b) one can be obtained from the other by swapping columns inside the column boundaries, and the rest of the table remains the same regardless of which row is selected first.

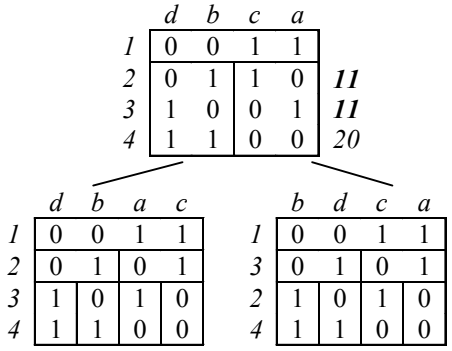

Figure 7. Example of row symmetry

The symmetry relation between rows is transitive, and rows can be divided into disjunctive symmetry sets.

In order to check if two rows are symmetric, the columns are swapped within the column boundaries to attempt to transform the first row into the second row. There could be different column orderings that convert one row into the other. All of them are valid, but some will make the rest of the table the same as the starting table and some will not. Two rows are not symmetric if there is no ordering of the columns that will transform the table into an identical one. This can be computationally intensive, so we propose a fast method to accomplish that.

Definition 4. Two columns are symmetric if the table remains invariant after they are swapped, with possible row reordering.

The column symmetry relation is an equivalence relation, and the columns can be divided into disjunctive symmetry sets. Symmetry between the columns reflects the symmetry of the Boolean function's inputs. In the example in Fig. 8, there are two symmetry sets: $C_1 = \{a, b\}$, and $C_2 = \{c, d, e\}$. The variables from the symmetry set can be interchanged, without changing the Boolean function. Column symmetry is preserved through all the iterations of the Boolean matching algorithm.

Group symmetry of variables is also utilized to detect equivalent branches in the algorithm. Group symmetry can exist between the members of symmetry sets with the same cardinality. In the example in Figure 6, variables *a* and *c*, and *b* and *d* are not symmetric, i.e. exchanging only one pair of these variables modifies the truth table. But swapping *both* of these pairs (*a*, *c*) and (*b*, *d*) simultaneously keeps the table invariant. This type of special symmetry is also referred to in [1] as *hierarchical symmetry.*

Definition 5. Two column symmetry sets belong to a column symmetry group if a) they have the same cardinality and b) after swapping the columns from the first set with the columns from the second set, the table stays invariant.

The column symmetry group relation is also transitive. Using the column symmetries, we can easily determine if two rows are symmetric. Since symmetry of columns is preserved throughout the algorithm, symmetry sets and groups are calculated only once.

The ones and the zeros in the minimum signature rows are labeled with their corresponding column symmetry sets. If the labels for the ones in both rows are the same, as well as the labels for the zeros, looking between the column boundaries,

then the rows are symmetric. The order of the labels within the column boundaries does not matter, because the columns can be swapped. In the example in Fig. 8, rows 2, 3, and 4 have the same minimum signature 11. Rows 2 and 3 belong to the same row symmetry set, and another symmetry set is row 4. The algorithm would branch using minimum rows 2 and 4. Eventually, the branch from row 4 will be stopped, because its minimum row signature is greater than the minimum row signature in the branch from row 2. Figure 9 presents the pseudo-code for symmetry check for the rows with the same minimum row signature.

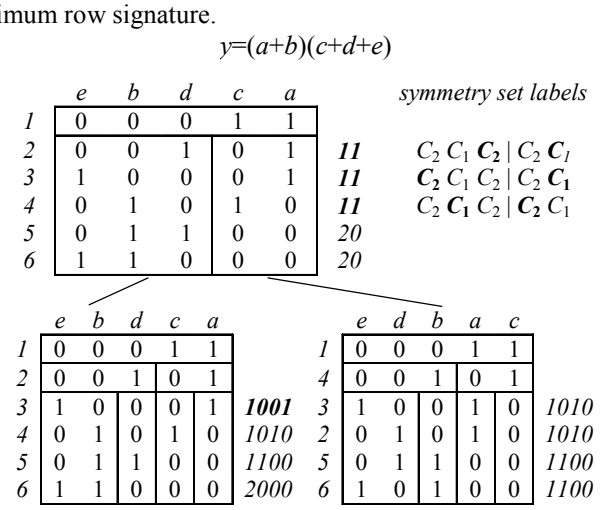

Figure 8. Using column symmetry to determine the symmetry of rows

```
obtain_column_symmetry_sets_and_groups();
for ∀i ∈ min_row_set {
  (set_label_0[i],set_label_1[i]) =
           calculate_symmetry_set_labels();
  (group_label_0[i],group_label_1[i]) =
        calculate_symmetry_group_labels();}
new_min_row_set ← r ∈ min_row_set;
for ∀i ∈ (min_row_set \ new_min_row_set){
 for ∀j ∈ new_min_row_set {
  if(set_label_0[i] != set_label_0[j] &&
        set_label_1[i] != set_label_1[j]) {
    if (group_label_0[i]!=group_label_0[j]
    && group_label_1[i]!=group_label_1[j]){
     new_min_row_set ← min_row_set[i];
    } } } }
min_row_set = new_min_row_set;
```

Figure 9. Pseudo-code for symmetry check

The symmetry check is performed after Step 2.1 in the *MCCF* algorithm if there are multiple rows with the same minimum signature and only non-symmetric rows are passed on to the next step.

5. Experimental Results

The MCCF algorithm was tested on a large set of CMOS gates with a constraint on the number of series (*s*) and parallel (*p*) transistors. The gates were constructed as AND-OR trees [8]. An AND-OR tree is a tree whose internal nodes have two or more children, and a node can be an AND node or an OR node. The AND and OR nodes alternate along each path from the root to a leaf. The leaves are labeled with an input variable. The root is labeled with a (*s*, *p*) constraint, where *s* is the maximum number of transistors in series and *p* is the maximum number of transistors in parallel. All the gates with a (*s*, *p*) constraint can be divided in two groups: and-or gates with an AND node as a root, and or-and gates with an OR node as a root. The Boolean function of an AND-OR tree is obtained by the in-order traversal of the tree. Due to space limitations we omit the algorithm for generating the AND-OR trees with a (*s*, *p*) constraint.

The Boolean equations of the AND-OR trees are converted into the implicant tables using SIS [9]. The Boolean functions representing the AND-OR trees are all positive unate. The implicant table is transformed into a table of ones and zeros, where the don't cares are assigned a value of zero. Table 1 presents the results of running the *MCCF* algorithm with symmetry check as described in section 4 on all of the gates with (*s*, *p*) limits up to (5, 5). The second column presents the total number of gates with the (*s*, *p*) constraint. The next columns are subdivided into the worst-case number for a function with the corresponding (*s*, *p*) limit, the average, and the median among all functions with that limit. The third column corresponds to the total number of nodes in the execution tree. The fourth column is the maximum number of tables in the linked list *LL* among all iterations of the *MCCF* algorithm. It represents the maximum width of the execution tree. The fifth columns is the CPU time in milliseconds on a Sun UltraSparc 60 workstation spent for the execution of the whole *MCCF* algorithm. The last column is a comparison with the Boolean matching method described in [6]. Essentially, it presents the number of permutations needed to determine input correspondence due to aliasing errors. The number of permutations is equal to $\prod_{i=1}^{q}(S_i!)$, where S_i is the cardinality of a symmetry class whose elements are symmetry sets with *i* elements [6].

It can be observed from Table 1 that the cut point where aliasing errors cause a larger number of permutations than the number of nodes in the execution tree of the *MCCF* algorithm occurs for (*s*, *p*) ≥ (3, 3). For libraries with small number of inputs and up to three transistors in series and/or parallel, Boolean matching methods based on signatures and BDDs have comparable complexity with the *MCCF* algorithm. But for large functions with lots of inputs, on average and in the worst case, the proposed *MCCF* algorithm outperforms previous Boolean matching methods.

In all tested (*s*, *p*) functions in Table 1, the *MCCF* algorithm with use of symmetries produced a single canonical form at the end. All equivalent branches were eliminated using the proposed method in section 4, which contributed to the fast and feasible run-time and computational complexity for large functions.

Table 1. Experimental results for the MCCF algorithm on a set of gates with a (s, p) limit

(s, p)	# of functions	MCCF total # of nodes in the execution tree			MCCF max width of the execution tree			MCCF CPU time [ms]		# of permutations in [6]		
		worst	average	median	worst	average	median	worst	average	worst	average	median
(2, 3)	18	10	4.37	4	2	1.16	1	< 1	< 1	6	1.95	2
(3, 2)	18	9	3.52	3	1	1.00	1	< 1	< 1	6	1.95	2
(3, 3)	87	28	6.43	5	3	1.25	1	10	0.263	24	4.32	2
(3, 4)	396	65	10.50	8	4	1.55	1	20	1.439	120	10.73	6
(4, 3)	396	92	8.79	6	6	1.27	1	40	1.228	120	10.73	6
(4, 4)	3503	257	15.88	10	6	1.62	1	280	3.994	720	35.34	12
(4, 5)	28435	626	25.41	15	7	1.90	2	1520	10.252	5040	117.96	24
(5, 4)	28435	1025	23.75	12	12	1.63	1	3910	11.208	5040	117.96	24
(5, 5)	425803	3126	41.27	20	12	1.94	2	37280	35.722	40320	466.34	48

The *MCCF* algorithm can be used for examples where the number of permutations due to aliasing errors is large, and where the Boolean function can be represented with an implicant table with a feasible number of rows. For example, the function $f = x_0x_1+x_2(x_3+x_4) +x_5x_6(x_7+x_8)+(x_9+x_{10})(x_{11}+x_{12})+ +x_{13}(x_{14}+x_{15})(x_{16}+x_{17})$ requires 8!2! = 80640 permutations to find the input correspondence using signatures in [6]. This function has 18 columns and 20 rows in the implicant table and it took 0.02s to find the canonical form. The total number of nodes in the execution tree was 20 and the maximum width of the execution tree was 3.

6. Conclusion

This paper presented a new algorithm for obtaining a canonical form of a Boolean function under any permutation of its inputs, where the function is represented in the form of a truth table or a minimized sum-of-products form. The minimum cost of the canonical form of the table is used as a signature for uniquely identifying the function in the library.

A branch-and-bound algorithm, with a very tight bounding function, is used to find the canonical form. Using symmetries of the input variables, the algorithm avoids equivalent branches and is able to handle very large functions. The algorithm can be used for obtaining a unique variable ordering in ROBDDs, so it can be incorporated with BDD representations of Boolean networks.

Experimental results on more than 100,000 gates (with up to 25 inputs and 3125 implicants) confirm the validity and feasible run-time of the algorithm. The best performance is achieved with large libraries and very complex Boolean functions, where other methods based on signatures and BDDs require large number of permutations for finding the right input correspondences.

7. Acknowledgments

This research was supported by grants from Semiconductor Research Corporation, National Science Foundation (NSF), the NSF Center for the Design of Analog and Digital ICs (CDADIC), Sun Microsystems and Intel Corporation. The authors acknowledge the help of Paul Tseng, Ted Stanion and Tyler Thorp.

8. References

[1] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent boolean comparison," *Proc. of ASP Design Automation Conference*, 1995, pp. 459-464.

[2] Q. Wu, C. Chen, and J. Acken, "Efficient boolean matching algorithm for cell libraries," *Proc. of Int. Conference on Computer Design*, October 1994, pp. 36-39.

[3] J. Burch and D. Long, "Efficient boolean function matching," *Proc. of Int. Conference on Computer-Aided Design*, 1992, pp. 408-411.

[4] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," *Proc. of Design Automation Conference*, June 1998, pp. 206-211.

[5] U. Schlichtmann and F. Brglez, "Efficient Boolean Matching in Technology Mapping with Very Large Cell Libraries," *Proc. of IEEE Custom Integrated Circuits Conference*, 1993, pp. 3.6.1-3.6.6.

[6] F. Mailhot and G. De Micheli, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 5, May 1993, pp. 599-620.

[7] L. Benini and G. De Micheli, "A Survey of Boolean Matching Techniques for Library Binding," *ACM*

Transactions on Design Automation of Electronic Systems, Vol. 2, No. 3, July 1997, pp. 193-226.

[8] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Technology Mapping in MIS," *Proc. of the Intl. Conf. of Computer Aided Design*, November 1987, pp. 116-119.

[9] E. Sentovich, et al., "SIS: A System for Sequential Circuit Synthesis," *Technical Report UCB/ERL M92/41*, University of California at Berkeley, May 1992.

[10] S. Devadas, A. Ghosh, and K. Keutzer, *Logic Synthesis*, McGraw-Hill Series on Computer Engineering, 1994.

[11] J. Mohnke and S. Malik, "Permutation and Phase Independent Boolean Comparison," *Integration, the VLSI Journal*, 16, December 1993, pp. 109-129.

9. Appendix

Proof of Lemma1: The proof is by induction. For $cr = 1$, we have only one table in the list, equal to the whole truth table. From step 1, all minimum rows $r \in min\_row\_set(A^1)$, have the same minimum cost $S_{min}{}^1$, equal to the number of ones in these rows. After permutation of columns in step 2, all ones are on the right in the row. The new column boundary is inserted at the column where the block of ones starts. Since all minimum rows r have the same number of ones, all new tables $A_r{}^1$ will have the same column boundary c_2.

Let's assume that after n iterations we have h column boundaries $c_1{}^i, c_2{}^i,..,c_h{}^i$. By iteration we mean the loop which begins at step 1 and ends at step 3. We assume that they are the same for all $A^i \in LL$. We want to prove that in the (n+1)-st iteration, the new inserted boundaries will be the same for all tables A^i in the list. Let's suppose that is not true, that for two tables $A_r{}^i$ and $A_p{}^j$ (with minimum rows r and p, respectively) the new column boundaries are different. From step 2.1, the row signatures for r and p are the same, $S_r{}^i = S_p{}^j = min\_signature$. It means that $s_{rl} = s_{pl}$, $\forall l = 1,..,h$ (step 1.1). The number of ones between the column boundaries (set in iteration n) is the same for rows r and p. In step 2.1.1, the columns in rows r and p are rearranged, such that ones are placed at the right side of the column boundaries. The new column boundaries are inserted before the columns where the blocks of ones start. If the new column boundaries are different for rows r and p, it means that the row signatures for r and p are different. That contradicts step 2.1.

Proof of Lemma2: All tables in the list have the same column boundaries (Lemma 1). For each minimum row, the row signatures are the same among all tables (step 2.1). The columns in the tables are rearranged in step 2.1.1 such that ones in the minimum rows are placed at the right side in each column boundary. Therefore, for each iteration of the algorithm, the minimized rows in all tables are identical.

Proof of Theorem 2: Let's assume that there is a table $A^j \notin LL$ that has smaller cost than all tables $A^i \in LL$. If A^j has the smallest cost, that can mean only one of the following: *(i)* the first row of table A^j has smaller cost than the first row of A^i; or *(ii)* the first n rows of A^j have the same cost as the first n rows of A^i, and the (n+1)-st row of A^j has smaller cost than the (n+1)-st row of A^i.

(i) For the first iteration ($cr = 1$), the column boundaries are set at the first row and at the end of the table. The row signature represents the number of ones in each row of the table. If the first row of table A^j has smaller cost than the first row of A^i, then the number of ones in the first row of table A^j is smaller than the number of ones in the first row of A^i. This is in contradiction with step 1.2 of the algorithm.

Using the claim from Lemma 2, after n iterations of the algorithm ($cr = n$) the cost of the top n rows of the tables A^i in the list LL are the same. Also, the column boundaries for all tables A^i are the same (Lemma 1). We assumed that the costs of the first n rows of tables A^i and A^j are the same. Because the cost represents a binary number, it follows that the top n rows are identical for tables A^i and A^j. Let's put column boundaries at each row of table A^j for the top n rows. The column boundaries are of the table A^j for the top n rows. If the top n rows are identical, then the column boundaries of A^i and A^j are the same too. If the cost of the (n+1)-st row of table A^j is smaller than the cost of the (n+1)-st row of table A^i, then its row signature is smaller too. That contradicts steps 1.2 and 2.1.