PAPER Special Section on VLSI Design and CAD Algorithms

Fast Boolean Matching under Permutation by Efficient Computation of Canonical Form

Debatosh DEBNATH[†a)], ***Nonmember and*** **Tsutomu SASAO**[††b)], ***Member***

SUMMARY Checking the equivalence of two Boolean functions under permutation of the variables is an important problem in the synthesis of multiplexer-based field-programmable gate arrays (FPGAs), and the problem is known as *Boolean matching*. This paper presents an efficient breadth-first search technique for computing a canonical form—namely *P-representative*—of Boolean functions under permutation of the variables. Two functions match if they have the same P-representative. On an ordinary workstation, on the average, the method requires several microseconds to check the Boolean matching of functions with up to eight variables against a library with tens of thousands of cells.

key words: *Boolean matching, technology mapping, variable permutation, P-equivalence*

1. Introduction

Boolean matching is a technique to detect the equivalence of two Boolean functions under permutation of the variables. One of the main application of Boolean matching is in technology mapping [10]. In a technology mapping environment, where Boolean matching of a large number of functions are required, a faster algorithm is desirable. Thus, efficient Boolean matching algorithms have been developed [1]. Boolean matching is also useful in logic verification where the correspondence of the inputs of the two circuits are unknown [4], [21], [26], [27] and in other areas of logic synthesis such as in the design of AND-OR-EXOR three-level networks [7].

In this paper we present an efficient Boolean matching algorithm, which has applications in the technology mapping of multiplexer-based field-programmable gate arrays (FPGAs) [2]. As a basis of the Boolean matching we use the P-representative, which is unique among the functions of a P-equivalence class. The set of functions that are equivalent under permutation of the variables form a *P-equivalence class* [13], [22]. In a P-equivalence class the function that has the smallest binary number representation is the *P-representative* of that class. Every P-equivalence class has a unique P-representative. Thus, if the P-representatives for the two functions are the same, one can be transformed into another by changing permutation of the variables. P-equivalence classes and P-representatives have been used in logic design for many years. Hellerman used them to show the catalog of minimal NAND and NOR networks for P-representative functions [14]. Harrison [13, pp.148–150] and Muroga [22, pp.327–332] provided detail technical and historical discussions on them. The paper is based on [8]. It uses a modified data structure. The present implementation is about 10% faster than the original implementation, but requires about 50% more memory. The original paper is modified by adding more introductory materials and references; new experimental results and comparison with another method are also added. The presentation is improved by adding new materials, which include three figures and an example.

To match against a library, our method works in two phases. First, it computes the P-representatives for all the elements in the library and stores them in a hash table during a *setup phase*. Second, it computes the P-representatives for the functions to be matched and checks the hash table for the same P-representatives during a *matching phase*. During the setup phase for multiplexer-based field-programmable gate arrays, it generates a library with all the cells that an FPGA module can implement by bridging the inputs and setting the inputs to constants. Important features of our method in relation to other methods are as follows:

- P-representative is a powerful notion because it is *unique* for any P-equivalence classes. Burch and Long introduced a semi-canonical form for matching under permutation of the variables [3]. However, semi-canonical form is non-unique. Recently, Hinsberger and Kolla [15], and Ciric and Sechen [5] developed Boolean matching methods based on the computation of canonical forms of Boolean functions. Wu et al. also proposed a canonical-form-based Boolean matching technique; but, the practical significance of the algorithm cannot be verified without implementation [28].
- As a basis of the Boolean matching many algorithms use *signatures*, which show some properties of the functions. Although signatures are extensively used in Boolean matching [18], [19], [23], they are unable to uniquely identify many P-equivalence classes. Thus, an exhaustive search is necessary to obtain a conclusive result. However, P-representative based method always gives a conclusive result without any exhaus-

Manuscript received March 22, 2004.
Manuscript revised June 16, 2004.
Final manuscript received August 5, 2004.

[†]The author is with the Department of Computer Science and Engineering, Oakland University, Rochester, Michigan 48309, U.S.A.
[††]The author is with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820-8502 Japan.
a) E-mail: debnath@oakland.edu
b) E-mail: sasao@cse.kyutech.ac.jp

tive search.

- Pairwise Boolean matching is not required in our method. Many Boolean matching methods require pairwise Boolean matching [16], [18], [26], [27]. Thus, they are unsuitable for handling libraries with a large number of cells, because pairwise Boolean matching of a function with the functions in a large library is time consuming.
- The computational complexity of our method is independent of the number of cells in the library, and it can efficiently handle libraries with extremely large number of cells. The number of cells is constrained only by the available memory resources. This feature is important in table look-up based logic synthesis [13], where matching against a library with more than one million cells is necessary [7]. Moreover, an increase in the number and in the size of the cells in a library improves the quality of the mapped circuits [17], [24], [25]. However, Boolean matching for large libraries is computationally expensive.
  On the other hand, our method efficiently deals with extremely large libraries. Libraries with a large number of cells are common in the technology mapping of FPGAs [24]. For example, the popular *ACT1 module* developed by Actel [11] generates a library with 702 cells [24]. Usually standard cell libraries contain far fewer cells than this number. For example, the *lib2* library from the MCNC, which is extensively used by the research community, contains only 27 cells [29].
- Cells with sufficiently large number of inputs can be handled by our method. The present implementation can treat cells with up to eight inputs; its practical upper limit is nine-input cells. For cells with more than nine inputs, the method requires gigabytes of memory. Our method can easily handle the largest cells generated from popular FPGAs. For example, the ACT1 module has eight inputs [11]. Therefore, Boolean matching for functions with only up to eight inputs is necessary when working with the ACT1.
- Our data structure for computing the P-representative is memory efficient. For up to seven-variable functions the method requires only about one megabyte memory. For functions with up to eight variables the memory requirement is about 15 megabytes.
- P-representative is simple and compact. Since cells with only up to several inputs are common in technology mapping of FPGAs, binary numbers are used as a compact and an efficient representation of Boolean functions that model the library cells. In this representation, the equivalence checking of a pair of functions is done by comparing integers.
  To represent all the cells generated from an ACT1 module, our method requires about five kilobytes of memory. We note that the ACT1 module generates total 702 cells whose average number of inputs is 4.77 [24]. On the other hand, to represent the ACT1 module for Boolean matching, some algorithms require more than two orders of magnitude higher memory than that of our method [12].
- Our method does not use any functional properties. It makes the method independent of any cell architecture and simplifies the programming task. Many Boolean matching algorithms heavily depends on functional properties to reduce the computation time [18], [19], [23], [26].
- Our method is fast and flexible. Experimental results show that it is more than one and two orders of magnitude faster than the method of Schlichtmann and Brglez [24] and that of Zhu and Wong [30], respectively. It can be used with *filters* [6], [18], [20] to further reduce the matching time.

The remainder of the paper is organized as follows: Sect. 2 introduces terminology. Section 3 develops the technique to compute the P-representative, which is the basis of our Boolean matching algorithm. Section 4 reports the experimental results. Section 5 presents conclusions.

2. Definitions and Terminology

This section defines the basic terminology that is necessary to explain the material in the paper.

Definition 1: The minterm expansion of an n-variable function is $f(x_1, x_2, \ldots, x_n) = c_0 \cdot \bar{x}_1 \bar{x}_2 \cdots \bar{x}_n \vee c_1 \cdot \bar{x}_1 \bar{x}_2 \cdots x_n \vee \cdots \vee c_{2^n-1} \cdot x_1 x_2 \cdots x_n$, where $c_0, c_1, \ldots, c_{2^n-1} \in \{0, 1\}$. The binary digit c_j is called the *coefficient of the j-th minterm*, *j-th coefficient*, or simply *coefficient*. The 2^n bit binary number $c_0 c_1 \cdots c_{2^n-1}$ is the *binary number representation* of f. To denote a binary number, a subscripted 2 is used after it.

Example 1: Consider the three-variable function $f(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee x_1$. The binary number representation of f is 10001111_2.

Logic functions can be grouped into classes by using simple transformations.

Definition 2: Two functions f and g are *P-equivalent* if g can be obtained from f by permutation of the variables [13], [22]. $f \overset{P}{\sim} g$ denotes that f and g are P-equivalent. P-equivalent functions form a *P-equivalence class* of functions.

Example 2: Consider the three functions: $f_1(x_1, x_2, x_3) = \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3$, $f_2(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_3 \vee x_1 x_2 x_3$, and $f_3(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \vee x_1 x_2 x_3$. Since $f_2(x_2, x_1, x_3) = \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3 = f_1(x_1, x_2, x_3)$, we have $f_1 \overset{P}{\sim} f_2$, and since $f_3(x_1, x_3, x_2) = \bar{x}_1 \bar{x}_3 \vee x_1 x_2 x_3 = f_2(x_1, x_2, x_3)$, we have $f_2 \overset{P}{\sim} f_3$. Therefore, the functions f_1, f_2, and f_3 belong to the same P-equivalence class.

Definition 3: The function that has the smallest binary number representation among the functions of a P-equivalence class is the *P-representative* of that class.

Example 3: All the functions of the P-equivalence class for $\bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3$ are $f_1(x_1, x_2, x_3) = \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3$,

$f_2(x_1, x_2, x_3) = \bar{x}_1\bar{x}_3 \vee x_1x_2x_3$, and $f_3(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2 \vee x_1x_2x_3$. In binary number representation: $\bar{x}_2\bar{x}_3 \vee x_1x_2x_3 = 10001001_2$, $\bar{x}_1\bar{x}_3 \vee x_1x_2x_3 = 10100001_2$, and $\bar{x}_1\bar{x}_2 \vee x_1x_2x_3 = 11000001_2$. Since $10001001_2 < 10100001_2 < 11000001_2$, the P-representative of this class is $\bar{x}_2\bar{x}_3 \vee x_1x_2x_3$.

For an n-variable function, there are at most $n!$ P-equivalents. Among them, our objective is to find the P-equivalent that has the smallest binary number representation as fast as possible.

3. Computing P-Representative

In this section, we show a method for computing P-representative, mainly, by using three- and four-variable functions. It can be easily extended to functions with more variables.

3.1 Naive Method

The truth-table for a three-variable function $f(x_1, x_2, x_3)$ is shown in Fig. 1(a), where $c_0, c_1, \ldots, c_7 \in \{0, 1\}$. We want to prepare the truth-table for $f(x_3, x_2, x_1)$ in Fig. 1(b). We do this by copying the coefficients in Fig. 1(a) to Fig. 1(b), such that $f(a, b, c)$ in Fig. 1(a) and $f(c, b, a)$ in Fig. 1(b) become the same, where $a, b, c \in \{0, 1\}$. The permutation of the variables for the functions in Figs. 1(a) and 1(b) are (x_1, x_2, x_3) and (x_3, x_2, x_1), respectively. Similarly, we can generate functions with other permutations of the variables, and take the function that has the smallest binary number representation as the P-representative.

A close observation to the coefficients in Fig. 1 reveals that most of the coefficients of $f(x_1, x_2, x_3)$ moved to new positions in $f(x_3, x_2, x_1)$. For example, the fifth coefficient, c_4, of $f(x_1, x_2, x_3)$ becomes the second coefficient of $f(x_3, x_2, x_1)$. Note that each time we want to change the permutation of the variables of an n-variable function, we must compute the new positions for all the 2^n coefficients. An n-variable function have at most $n!$ P-equivalents. Thus, to compute the P-representative for an n-variable function we must compute $n!2^n$ new positions for the coefficients. As a result, the method is computationally expensive.

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	c_0
0	0	1	c_1
0	1	0	c_2
0	1	1	c_3
1	0	0	c_4
1	0	1	c_5
1	1	0	c_6
1	1	1	c_7

(a) Truth-table for $f(x_1, x_2, x_3)$.

x_3	x_2	x_1	$f(x_3, x_2, x_1)$
0	0	0	c_0
0	0	1	c_4
0	1	0	c_2
0	1	1	c_6
1	0	0	c_1
1	0	1	c_5
1	1	0	c_3
1	1	1	c_7

(b) Truth-table for $f(x_3, x_2, x_1)$.

Fig. 1 Two different permutations of the variables of a three-variable function.

3.2 Using Precomputed New Coefficient Positions

The variables of $f(x_1, x_2, x_3)$ can be permuted in six ways: (x_1, x_2, x_3), (x_1, x_3, x_2), (x_2, x_1, x_3), (x_2, x_3, x_1), (x_3, x_1, x_2), and (x_3, x_2, x_1). Figure 2 shows a three-variable function $f(x_1, x_2, x_3)$ and its all possible P-equivalents. We can say that Fig. 2 shows the *new coefficient positions* which can be used to generate P-equivalents. Thus, by using the precomputed new coefficient positions in Fig. 2, we can easily generate all the P-equivalents of any given three-variable function. This method is much faster than the naive method of Sect. 3.1, because computation of the new positions for the coefficients is unnecessary. Figure 3 shows all the P-equivalents of a four-variable function; it is similar to Fig. 2 except column headings are removed and c_j is replaced by j $(0 \leq j \leq 15)$. Although column headings are removed from Fig. 3 for ease of showing the whole table, they are required by our algorithm.

3.3 Using Breadth-First Search

For an n-variable function, the above method first generates $n!$ functions and then chooses the function that has the smallest binary number representation as the P-representative. Since we are interested only in the function that has the smallest binary number representation, we use breadth-first search technique for early detection of the variable permutation that cannot lead to the P-representative. We discard the variable permutation from consideration as soon as we detect that it cannot lead to the P-representative. The breadth-first search technique is difficult to apply without Fig. 2.

3.4 More Efficient Method

The above method uses breadth-first search on the new coefficient positions in Fig. 2. The method is fast; but, we can further speed-up the computation. For all the functions in

$f(x_1, x_2, x_3)$	$f(x_1, x_3, x_2)$	$f(x_2, x_1, x_3)$	$f(x_2, x_3, x_1)$	$f(x_3, x_1, x_2)$	$f(x_3, x_2, x_1)$
c_0	c_0	c_0	c_0	c_0	c_0
c_1	c_2	c_1	c_4	c_2	c_4
c_2	c_1	c_4	c_1	c_4	c_2
c_3	c_3	c_5	c_5	c_6	c_6
c_4	c_4	c_2	c_2	c_1	c_1
c_5	c_6	c_3	c_6	c_3	c_5
c_6	c_5	c_6	c_3	c_5	c_3
c_7	c_7	c_7	c_7	c_7	c_7

Fig. 2 All possible P-equivalents of a three-variable function $f(x_1, x_2, x_3)$.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	2	2	2	2	2	2	4	4	4	4	4	4	8	8	8	8	8	8
2	2	4	4	8	8	1	1	4	4	8	8	1	1	2	2	8	8	1	1	2	2	4	4
3	3	5	5	9	9	3	3	6	6	10	10	5	5	6	6	12	12	9	9	10	10	12	12
4	8	2	8	2	4	4	8	1	8	1	4	2	8	1	8	1	2	2	4	1	4	1	2
5	9	3	9	3	5	6	10	3	10	3	6	6	12	5	12	5	6	10	12	9	12	9	10
6	10	6	12	10	12	5	9	5	12	9	12	3	9	3	10	9	10	3	5	3	6	5	6
7	11	7	13	11	13	7	11	7	14	11	14	7	13	7	14	13	14	11	13	11	14	13	14
8	4	8	2	4	2	8	4	8	1	4	1	8	2	8	1	2	1	4	2	4	1	2	1
9	5	9	3	5	3	10	6	10	3	6	3	12	6	12	5	6	5	12	10	12	9	10	9
10	6	12	6	12	10	9	5	12	5	12	9	9	3	10	3	10	9	5	3	6	3	6	5
11	7	13	7	13	11	11	7	14	7	14	11	13	7	14	7	14	13	13	11	14	11	14	13
12	12	10	10	6	6	12	12	9	9	5	5	10	10	9	9	3	3	6	6	5	5	3	3
13	13	11	11	7	7	14	14	11	11	7	7	14	14	13	13	7	7	14	14	13	13	11	11
14	14	14	14	14	14	13	13	13	13	13	13	11	11	11	11	11	11	7	7	7	7	7	7
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Fig. 3 All possible P-equivalents of a four-variable function (only j is shown for c_j).

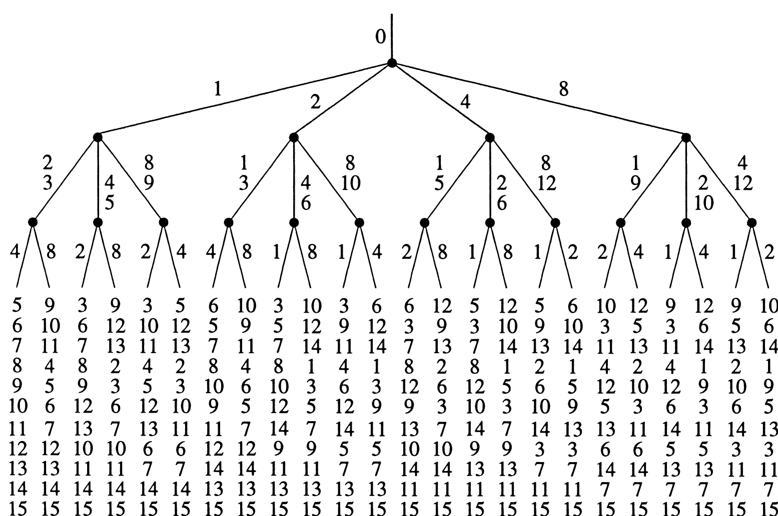

Fig. 4 Breadth-first search tree for four variables.

Fig. 2 the first coefficients are the same. Thus, in breadth-first search any comparison is unnecessary for these coefficients. Next we consider the second coefficients for all the functions in Fig. 2. The usual way is to generate all the second coefficients, and then retain only the variable permutations that have the smallest value for the second coefficients and discard other variable permutations. However, if we partition all the variable permutations in Fig. 2 into three sets, $\{(x_1, x_2, x_3), (x_2, x_1, x_3)\}$, $\{(x_1, x_3, x_2), (x_3, x_1, x_2)\}$, $\{(x_2, x_3, x_1), (x_3, x_2, x_1)\}$, then instead of generating all the second coefficients we need to generate only three coefficients, because for each of these sets the second coefficients are the same. We retain only the variable permutations that have the smallest value for the second coefficients and discard other variable permutations. Then the search continues with the third coefficients in Fig. 2. Thus, we can reduce the computation time for the second coefficients by a factor of 2 (= $3!/3$). By using this technique for the n-variable functions, we can reduce the computation time for the second coefficients by a factor of $n!/n$. For functions with more than three variables this technique is very effective to reduce the computation time, because we can recursively partition the variable permutations. In general, for n-variable functions, we can partition the variable permutations into n sets at first, then each of these sets can be again partitioned into $n-1$ sets; we can recursively partition each of these sets until the cardinality of the sets become one. As a result, we can reduce the computation time for many other coefficients.

We incorporate this idea to find the P-representative as the traversal of a *breadth-first search tree*, which also uses the new coefficient positions in Fig. 2. During the *setup phase* of the Boolean matching we build this tree, which is the main data structure of our algorithm.

The P-representatives in Fig. 3 are arranged such that they can be partitioned into four sets where the second coefficients in each set are the same and that the P-representatives in each set occupy contiguous positions. In a similar manner we arrange the P-representatives in each of these sets such that they can be again partitioned into three sets where the third coefficients in each set are the same and that the P-representatives in each set occupy contiguous positions. We recursively partition each of these sets until the cardinality of the sets become 1. The breadth-first search tree for four variables is shown in Fig. 4, which is built ac-

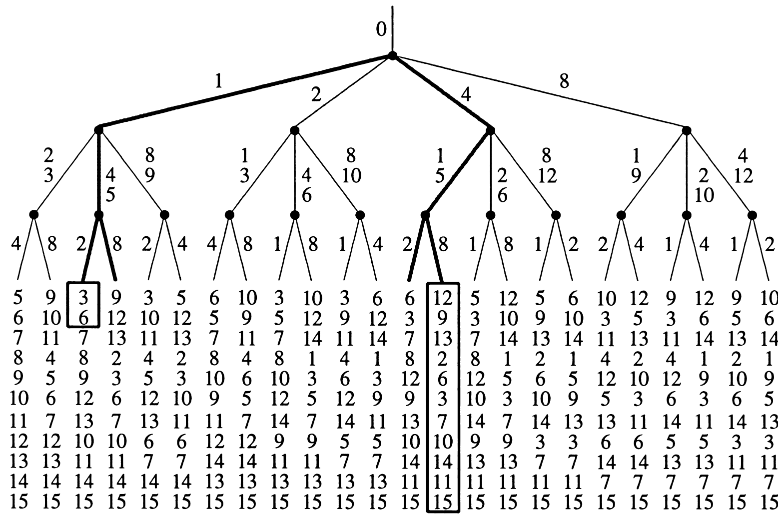

Fig. 5 Search paths for the four-variable function 1010011111110011_2.

cording to the partitions outlined above. Similar observations are used to build search trees for more variables.

Figure 5 shows an example to find the P-representative for the four-variable function 1010011111110011_2 (i.e., the coefficients $c_1 = c_3 = c_4 = c_{12} = c_{13} = 0$ and $c_0 = c_2 = c_5 = c_6 = c_7 = c_8 = c_9 = c_{10} = c_{11} = c_{14} = c_{15} = 1$). Since we are only interested in finding the function with the smallest binary number representation, at the top level of the tree we discard branches for c_2 and c_8 because these coefficients have a value 1 and the other coefficients have a value 0. Thus, the paths for c_1 and c_4 are selected. In the next level, in a similar manner, we select only two branches. We continue this process until we reach the leaves of the tree. The branches that we traverse to find the P-representative are shown in the thick lines or small rectangles. In this way, we need to search only a small portion of the tree to find the P-representative. From Fig. 5 the P-representative for the given function is 1001101011011111_2.

Figure 5 shows that each node of the breadth-first search tree has multiple children, and that the number of children of a node depends on the level of the node and on the number of variables. For n-variable functions, the tree has n levels. Let the root node of the tree is at level 1. Thus, the leaf nodes are at level n. The number of children of a node at level k is $n - k + 1$, where $1 \leq k < n$. Thus, the total number of nodes in the tree is $1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n(n-1)(n-2) \cdots 4 \cdot 3 \cdot 2$.

The new coefficient positions in Fig. 2 shows that the first coefficients are the same for all the functions. This is also true for the last coefficients. Figure 2 also shows that, for $n = 3$ if the i-th coefficient is c_k, then the $(2^n - i + 1)$-th coefficient is $c_{2^n - k - 1}$ where $1 \leq i \leq 2^{n-1}$. We found these properties are true for $2 \leq n \leq 9$ and conjecture they are true for any arbitrary n. These properties can be used to save memory resources for functions with many variables, where memory consumption is a crucial issue.

4. Experimental Results

We implemented the proposed method of Boolean matching for functions with up to eight variables on a Sun Fire 280R Server (900-MHz UltraSPARC-III CPU). It consists of about 3,000 lines of C code and about 11 megabytes of dynamically linked data for the new coefficient positions. The program requires about 15 megabytes memory, most of which is used for the matching of functions with eight variables. If the program is used for the functions with up to seven variables, it needs only about one megabyte memory. We note that additional memory is required to store P-representatives of the library cells. During the setup phase the program constructs the breadth-first search trees; it takes about 30 milliseconds.

To demonstrate the effectiveness of our matching technique, we conducted an experiment by using 5,000,000 pseudo-random functions with three to eight variables and tried to match them against a library with 100,000 randomly generated cells. We computed the P-representatives for all the library cells and stored them in a hash table during the setup phase. Then the P-representatives for each of the pseudo-random functions are computed and compared with the P-representatives for the library cells in the hash table. Table 1 summarizes the average Boolean matching time in microseconds; it is the time to match a function against the entire library cells.

Schlichtmann and Brglez reported that, for three-, four-, five-, six-, seven-, and eight-variable functions the Boolean matching time is approximately 2.0, 4.0, 6.0, 8.0, 12.0, and 18.0 milliseconds, respectively, on a DECStation 5000/200 (25-MHz MIPS R3000 CPU) [24]. Thus, even considering a Sun Fire 280R Server is about 100 times faster than a DECStation 5000/200, our method is more than an order of magnitude faster than the method of Schlichtmann and Brglez.

Table 1 Average time for Boolean matching against a library with 100,000 cells.

Number of Variables	Time (microseconds)
3	1.09
4	2.52
5	3.69
6	4.71
7	7.03
8	11.27

Zhu and Wong reported that to check the Boolean matching of all the four-variable functions against the ACT1 FPGA module requires 72 minutes on a Sun SPARCstation 1 (20-MHz SuperSPARC CPU), i.e., average matching time for four-variable case is 65.92 milliseconds per function [30]. An ACT1 module has eight inputs, and 702 different cells can be generated by bridging its inputs and setting its inputs to constants [24]. To compare our method with Zhu and Wong's method, we assume that cell generation takes as much time as Boolean matching takes. Thus, considering a Sun Fire 280R Server is about 100 times faster than a Sun SPARCstation 1, our method is more than two orders of magnitude faster than the method of Zhu and Wong.

5. Concluding Remarks

In this paper we used the notion P-representative, which is unique for any P-equivalence classes, and presented a breadth-first search algorithm for its quick computation. We demonstrated the usefulness of P-representatives for efficient Boolean matching against a large library. The concept P-representative is extended to NP- and NPN-equivalence classes [13], [22], and a breadth-first search approach is also devised to compute the representatives. The preliminary results are promising [9]. Our method is fast and flexible; it can be used with filters [6], [18], [20] to further reduce the computation time. To the best of our knowledge, we are unaware of any Boolean matching methods that can handle libraries with tens of thousands of elements and have a comparable speed performance. Our future work includes extension of the proposed method to functions with more variables and development of a technology mapping system by using the proposed matching technique.

Acknowledgement

This work was supported in part by the Japan Society for the Promotion of Science and in part by the Ministry of Education, Science, Culture, and Sports of Japan.

References

[1] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," ACM Trans. Des. Autom. Electron. Syst., vol.2, no.3, pp.193–226, July 1997.

[2] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic, Field-Programmable Gate Arrays, Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[3] J.R. Burch and D.E. Long, "Efficient Boolean function matching," Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp.408–411, Nov. 1992.

[4] D.I. Cheng and M. Marek-Sadowska, "Verifying equivalence of functions with unknown input correspondence," Proc. European Conf. on Design Automation, pp.81–85, Feb. 1993.

[5] J. Ciric and C. Sechen, "Efficient canonical form for Boolean matching of complex functions in large libraries," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.22, no.5, pp.535–544, May 2003.

[6] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," Formal Methods in System Design: An Int. Journal, vol.10, no.2, pp.137–148, April 1997.

[7] D. Debnath and T. Sasao, "A heuristic algorithm to design AND-OR-EXOR three-level networks," Proc. Asia and South Pacific Design Automation Conf., pp.69–74, Feb. 1998.

[8] D. Debnath and T. Sasao, "Fast Boolean matching under permutation using representative," Proc. Asia and South Pacific Design Automation Conf., pp.359–362, Jan. 1999.

[9] D. Debnath and T. Sasao, "Efficient computation of canonical form for Boolean matching in large libraries," Proc. Asia and South Pacific Design Automation Conf., pp.591–596, Jan. 2004.

[10] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, New York, 1994.

[11] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K.A. El-Ayat, and A. Mohsen, "An architecture for electrically configurable gate arrays," IEEE J. Solid-State Circuits, vol.24, no.2, pp.394–398, April 1989.

[12] S. Ercolani and G. De Micheli, "Technology mapping for electrically programmable gate arrays," Proc. IEEE/ACM Design Automation Conf., pp.234–239, June 1991.

[13] M.A. Harrison, Introduction to Switching and Automata Theory, McGraw-Hill, New York, 1965.

[14] L. Hellerman, "A catalog of three-variable OR-invert and AND-invert logical circuits," IEEE Trans. Electron. Comput., vol.EC-12, no.3, pp.198–223, June 1963.

[15] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," Proc. IEEE/ACM Design Automation Conf., pp.206–211, June 1998.

[16] M. Hütter and M. Scheppler, "Memory efficient and fast Boolean matching for large functions using rectangle representation," IEEE/ACM 12th Int. Workshop on Logic and Synthesis, May 2003.

[17] K. Keutzer, K. Kolwicz, and M. Lega, "Impact of library size on the quality of automated synthesis," Proc. IEEE/ACM Int. Conf. on Computer-Aided Design, pp.120–123, Nov. 1987.

[18] Y.-T. Lai, S. Sastry, and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," Proc. IEEE Int. Conf. on Computer Design, pp.452–458, Oct. 1992.

[19] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.12, no.5, pp.599–620, May 1993.

[20] Y. Matsunaga, "A new algorithm for Boolean matching utilizing structural information," IEICE Trans. Inf. & Syst., vol.E78-D, no.3, pp.219–223, March 1995.

[21] J. Mohnke, P. Molitor, and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," Formal Methods in System Design: An Int. Journal, vol.21, no.2, pp.167–191, Sept. 2002.

[22] S. Muroga, Logic Design and Switching Theory, John Wiley & Sons, New York, 1979.

[23] U. Schlichtmann, F. Brglez, and M. Hermann, "Characterization of Boolean functions for rapid matching in EPGA technology mapping," Proc. IEEE/ACM Design Automation Conf., pp.374–379, June 1992.

[24] U. Schlichtmann and F. Brglez, "Efficient Boolean matching in technology mapping with very large cell libraries," Proc. IEEE Custom Integrated Circuits Conf., pp.3.6.1–3.6.6, May 1993.
[25] V. Tiwari, P. Ashar, and S. Malik, "Technology mapping for low power in logic synthesis," Integration: The VLSI Journal, vol.20, no.3, pp.243–268, July 1996.
[26] C. Tsai and M. Marek-Sadowska, "Boolean functions classification via fixed polarity Reed-Muller forms," IEEE Trans. Comput., vol.46, no.2, pp.173–186, Feb. 1997.
[27] K.-H. Wang, T. Hwang, and C. Chen, "Exploiting communication complexity for Boolean matching," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.15, no.10, pp.1249–1256, Oct. 1996.
[28] Q. Wu, C.Y.R. Chen, and J.M. Acken, "Efficient Boolean matching algorithm for cell libraries," Proc. IEEE Int. Conf. on Computer Design, pp.36–39, Oct. 1994.
[29] S. Yang, Logic Synthesis and Optimization Benchmarks User Guide (Version 3.0), Technical Report, Microelectronics Center of North Carolina (MCNC), Jan. 1991.
[30] K. Zhu and D.F. Wong, "Fast Boolean matching for field-programmable gate arrays," Proc. IEEE European Design Automation Conf., pp.352–357, Sept. 1993.

Debatosh Debnath received the B.Sc.Eng. and M.Sc.Eng. degrees from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 1991 and 1993, respectively, and the Ph.D. degree from the Kyushu Institute of Technology, Iizuka, Japan, in 1998. He held research positions at the Kyushu Institute of Technology from 1998 to 1999 and at the University of Toronto, Ontario, Canada, from 1999 to 2002. In 2002, he joined the Department of Computer Science and Engineering at the Oakland University, Rochester, Michigan, as an Assistant Professor. His research interests include logic synthesis, design for testability, multiple-valued logic, and CAD for field-programmable devices. He was a recipient of the Japan Society for the Promotion of Science Postdoctoral Fellowship.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a Professor of the Department of Computer Science and Electronics at the Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including *Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis*, and *Logic Synthesis and Verification*, Kluwer Academic Publishers, 1993, 1996, 1999, and 2001, respectively. He has served as Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He received the NIWA Memorial Award in 1979, Distinctive Contribution Awards from the IEEE Computer Society MVL-TC in 1987, 1996, and 2004 for papers presented at ISMVLs, and Takeda Techno-Entrepreneurship Award in 2001. He has served as an Associate Editor of the *IEEE Transactions on Computers*. He is a fellow of the IEEE.