PAPER Special Section on VLSI Design and CAD Algorithms

Efficient Computation of Canonical Form under Variable Permutation and Negation for Boolean Matching in Large Libraries

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

SUMMARY This paper presents an efficient technique for solving a Boolean matching problem in cell-library binding, where the number of cells in the library is large. As a basis of the Boolean matching, we use the notion NP-representative (NPR): two functions have the same NPR if one can be obtained from the other by a permutation and/or complementation(s) of the variables. By using a table look-up and a tree-based breadth-first search strategy, our method quickly computes the NPR for a given function. Boolean matching of the given function against the whole library is determined by checking the presence of its NPR in a hash table, which stores NPRs for all the library functions and their complements. The effectiveness of our method is demonstrated through experimental results, which show that it is more than two orders of magnitude faster than the Hinsberger-Kolla's algorithm.

key words: *logic synthesis, Boolean matching, cell-library binding, technology mapping, canonical form*

1. Introduction

Determining whether a circuit is functionally equivalent to another under a permutation of its inputs, complementation of its one or more inputs, and/or inversion of its output is an important problem in logic synthesis, and *Boolean matching* technique is used to solve it. Algorithms for Boolean matching have applications in cell-library binding where it is necessary to repeatedly check if some part of a multiple-level representation of a Boolean function can be realized by any of the cells from a given library [9], in logic verification where correspondence of the inputs of two circuits are unknown [18], [30], and in table look-up based logic synthesis [7]. In this paper, we consider Boolean matching problem for cell-library binding. An exhaustive method for Boolean matching is computationally expensive even for functions with only few variables, because the time complexity of such an algorithm for an n-variable function is $O(n!2^{2n})$.

Boolean matching phase is a time consuming steps in cell library binding. Because of their importance in synthesizing cost effective circuits, Boolean matching problems received much attention and many algorithms have been developed to efficiently solve them [2]. Signatures, which are computed from some properties of Boolean functions, are extensively used in Boolean matching [2]. An equality in the signatures is a necessary condition for Boolean matching of two functions; although it is not the sufficient condition, signature-based algorithms have successfully demonstrated their effectiveness. Some of the signature-based algorithms are efficient for performing pair-wise Boolean matching [18], [22], [29], [30]. However, to match a function against a library they often require to perform pair-wise matchings of the function with all the library cells. Therefore, Boolean matching techniques based on them are unsuitable for handling libraries with large number of cells. There are other signature-based algorithms that are successfully used with cell libraries; however, they can handle libraries with only modest size [6], [20], [27]. Moreover, due to the lack of sufficient information in the signatures, algorithms based on them in many cases are unable to conclude a Boolean match. Thus, an exhaustive search is necessary to obtain a conclusive result. Some other Boolean matching algorithms consider only some restricted form of Boolean matching [10], [14], [25], [26].

There are other categories of Boolean matching algorithms that are based on the computation of some canonical form for Boolean functions [4], [5], [8], [12], [31]. Two functions match if their canonical forms are the same. The Boolean matching technique that we consider in this paper falls under this category. Burch and Long introduced a canonical form for matching under complementation and a semi-canonical form for matching under permutation of the variables [4]. These two forms can be combined to check Boolean matching under permutation and complementation of variables. However, a large number of forms for each cells are required when using the method in cell library binding. Ciric and Sechen [5], Debnath and Sasao [8], and Wu et al. [31] also proposed canonical forms for efficient Boolean matching. However, these techniques are applicable for Boolean matching under permutation of the variables only. Hinsberger and Kolla introduced a canonical form for solving the general Boolean matching problem that we are considering in this paper [12]. It can handle libraries with large number of cells under permutation and complementation of the variables as well as inversion of the function; however, the method requires considerable computation.

Manuscript received March 15, 2006.
Final manuscript received July 12, 2006.

[†]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

DOI: 10.1093/ietfec/e89–a.12.3443

In this paper, we present an efficient technique for computing a canonical form for Boolean functions. The canonical form — which we refer to as *NP-representative* (*NPR*) — remains unchanged under permutation and complementation of the variables. The set of functions that can be made identical under permutation and complementation of the variables form an *NP-equivalence class* [11], [13], [23]. In an NP-equivalence class, the function that has the smallest value in the binary representation is the NPR of the class, and every NP-equivalence class has a unique NPR. Thus, if the NPRs for the functions represented by the two circuits are equal, they are functionally equivalent under the permutation and complementation of inputs. It should be noted that our canonical form is similar to that of Hinsberger and Kolla [12]. For efficient computation of NPRs we use precomputed *NP-transformation tables* (*NPTTs*), which are used to quickly generate any functions in an NP-equivalence class. To make the search even more efficient, our method combines an NPTT with a search tree for each variables and performs breadth-first searches.

Although NPRs can identify functional equivalence of two circuits under permutation and complementation of inputs, they are unable to directly ascertain the functional equivalence if it involves determining whether the output of one circuit is also complemented. Thus, to handle the output complementation we use the following strategy. Let f and g be the Boolean functions represented by two circuits F and G, respectively. To check if F and G are functionally equivalent under permutation and complementation of inputs of G as well as possible complementation of its output, we compute the NPRs for f, g, and $\bar{g}$. If the NPRs for f and g are equal, G can be made functionally equivalent to F by permutation and complementation of inputs of G; however, if the NPRs for f and $\bar{g}$ are equal, complementation of the output of G is also required in addition to permutation and complementation of its inputs to make F and G functionally equivalent. In the case of a linear function [24], f and g can be equivalent under the complementation of the input. At the same time, f and g can be equivalent under the complementation of the output.

For using our method in cell library binding the library requires to be preprocessed. A set of personalized modules can be obtained from each library cells by bridging some of its inputs and/or setting some of its inputs to constant values [12]. In the preprocessing phase we generate *library functions*, which are the collection of functions represented by the library cells and by the personalized modules obtained from the library cells.

For Boolean matching of cell libraries we precompute the NPRs for the library functions and their complements. When we require to find a Boolean match of a given function against the whole library, we compute its NPR and check whether the same NPR is present in the precomputed NPRs for the library functions. An affirmative answer indicates a Boolean matching with a cell in the library. For efficient equivalence checking of NPRs we use a hash table to store the NPRs for the library functions.

Based on the above discussions, our Boolean matching technique for library binding can be summarized as:

- Build the search tree for each variable ($n = 3, 4, 5, 6,$ and 7).
- Generate the library functions; for each of them, compute two NPRs — one for it and the other for its complement — and store them in a hash table.
- Compute the NPR for the function to be matched against the library.
- Check the hash table for the presence of the NPR; a matching is found if the NPR is in the table.

We will refer to the first two of the above steps as the *setup phase* and the last two steps as the *matching phase*.

Usually Boolean matching for libraries with large number of cells is computationally expensive. However, an increase in the number of the cells in a library often helps optimize the area of the implementation [16], [19], [26]. Inclusion of complex cells to the library also improves other cost metrics of the resulting circuit. For example, for a set of 10 benchmark circuits, Tiwari et al. showed that by increasing the number of cells in a library from 33 to 396 reduces the power consumption of the mapped circuits by about 20 percent [28, p.266]. Kantabutra also presented strong statistical evidence in support of using complex cells [15]. However, inclusion of complex cells results in a large library. Traditionally, technology libraries are small. For example, Benini et al. used an industrial library containing 75 cells with up to five inputs [3], and *lib2* library — which is extensively used by the research community — consists of only 27 cells with up to six inputs [32].

Since pair-wise matchings are unnecessary, the computational complexity of our Boolean matching technique 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 [11], where matching against a library with more than one million cells is necessary [7]. Moreover, our method is independent of any cell architecture and any functional properties. However, functional properties can be used with our method as *filters* for quickly detecting the functions that cannot be matched against a library [6], [18], [20], [21], [25], [27]. Experimental results and comparison with another method demonstrate that the proposed technique is highly effective.

The remainder of the paper is organized as follows: Section 2 formally introduces the terminology. Section 3 develops the techniques for computing the NPR, which is the basis of our Boolean matching technique. Section 4 reports experimental results and compares our technique with another method. Section 5 presents conclusions.

2. Definitions and Terminology

Definition 1: Let the ***minterm expansion*** of an n-variable function $f(x_1, x_2, \ldots, x_n)$ be $m_0 \cdot \bar{x}_1 \bar{x}_2 \cdots \bar{x}_n \vee m_1 \cdot \bar{x}_1 \bar{x}_2 \cdots x_n \vee$

$\cdots \vee m_{2^n-1} \cdot x_1 x_2 \cdots x_n$, where $m_0, m_1, \ldots, m_{2^n-1} \in \{0, 1\}$. Let the 2^n bit binary number $m_0 m_1 \cdots m_{2^n-1}$, which is obtained by the concatenation of $m_0, m_1, \ldots, m_{2^n-1}$ in this order, be the ***binary representation*** of f. To denote a binary number, usually a subscripted 2 is used after it.

Example 1: Consider the three-variable function $f(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3$. The binary representation of f is $\mathtt{00011000}_2$.

Definition 2: Two functions f and g are ***NP-equivalent*** if g can be obtained from f by a permutation of the variables and/or complementation of one or more variables [11], [13], [23]. NP-equivalent functions form an ***NP-equivalence class*** of functions.

Example 2: Consider the four functions: $f_1(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3$, $f_2(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3$, $f_3(x_1, x_2, x_3) = \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3$, and $f_4(x_1, x_2, x_3) = \bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3$. Since $f_2(x_1, x_2, \bar{x}_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3 = f_1(x_1, x_2, x_3)$, f_1 and f_2 are NP-equivalent. Similarly, we can show that f_3 and f_4 are also NP-equivalent to f_1. Therefore, the functions f_1, f_2, f_3, and f_4 belong to the same NP-equivalence class.

Definition 3: The function that has the smallest value in the binary representation among the functions of an NP-equivalence class is the ***NP-representative (NPR)*** of that class.

Example 3: From Example 2, all the functions of an NP-equivalence class are $\bar{x}_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3$, $\bar{x}_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3$, $\bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3$, and $\bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3$. In the binary representation: $\bar{x}_1 \bar{x}_2 \bar{x}_3 \vee x_1 x_2 x_3 = \mathtt{10000001}_2$, $\bar{x}_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3 = \mathtt{01000010}_2$, $\bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 = \mathtt{00100100}_2$, and $\bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3 = \mathtt{00011000}_2$. Since $\mathtt{00011000}_2 < \mathtt{00100100}_2 < \mathtt{01000010}_2 < \mathtt{10000001}_2$, the NP-representative of the class is $\bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3$.

Variables of an n-variable function can be permuted in $n!$ ways and complemented in 2^n ways; thus the total number of possible combinations are $n!2^n$. However, for many functions some of these combinations generate the same function. Therefore, for an n-variable function there are at most $n!2^n$ NP-equivalents. Among them, our objective is to quickly find the NP-equivalent that has the smallest value in the binary representation.

3. Computing NP-Representative

In this section, we show a method to compute NP-representative (NPR) by using three-variable functions and discuss how the technique can be extended to functions with more variables.

3.1 Binary Representations under Permutation and Complementation of Variables

Binary representations of a given function under different permutation and complementation of variables can be easily generated if the function is represented as the minterm expansion. For example, let

$$\begin{aligned} f(x_1, x_2, x_3) = {} & m_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee m_1 \bar{x}_1 \bar{x}_2 x_3 \vee \\ & m_2 \bar{x}_1 x_2 \bar{x}_3 \vee m_3 \bar{x}_1 x_2 x_3 \vee \\ & m_4 x_1 \bar{x}_2 \bar{x}_3 \vee m_5 x_1 \bar{x}_2 x_3 \vee \\ & m_6 x_1 x_2 \bar{x}_3 \vee m_7 x_1 x_2 x_3 \end{aligned} \tag{1}$$

be the minterm expansion of a three-variable function, where $m_0, m_1, \ldots, m_7 \in \{0, 1\}$. The permutation of the variables in Eq. (1) is (x_1, x_2, x_3). When the permutation of the variables is (x_3, x_2, x_1), we have

$$\begin{aligned} f(x_3, x_2, x_1) = {} & m_0 \bar{x}_3 \bar{x}_2 \bar{x}_1 \vee m_1 \bar{x}_3 \bar{x}_2 x_1 \vee \\ & m_2 \bar{x}_3 x_2 \bar{x}_1 \vee m_3 \bar{x}_3 x_2 x_1 \vee \\ & m_4 x_3 \bar{x}_2 \bar{x}_1 \vee m_5 x_3 \bar{x}_2 x_1 \vee \\ & m_6 x_3 x_2 \bar{x}_1 \vee m_7 x_3 x_2 x_1 \\ = {} & m_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee m_4 \bar{x}_1 \bar{x}_2 x_3 \vee \\ & m_2 \bar{x}_1 x_2 \bar{x}_3 \vee m_6 \bar{x}_1 x_2 x_3 \vee \\ & m_1 x_1 \bar{x}_2 \bar{x}_3 \vee m_5 x_1 \bar{x}_2 x_3 \vee \\ & m_3 x_1 x_2 \bar{x}_3 \vee m_7 x_1 x_2 x_3. \end{aligned} \tag{2}$$

From Eqs. (1) and (2), the binary representations of $f(x_1, x_2, x_3)$ and $f(x_3, x_2, x_1)$ can be written as $m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7$ and $m_0 m_4 m_2 m_6 m_1 m_5 m_3 m_7$, respectively.

When variables are complemented, the binary representation can also be generated in a similar manner. For example, by replacing x_1 by $\bar{x}_1$ in Eq. (2), we have

$$\begin{aligned} f(x_3, x_2, \bar{x}_1) = {} & m_0 x_1 \bar{x}_2 \bar{x}_3 \vee m_4 x_1 \bar{x}_2 x_3 \vee \\ & m_2 x_1 x_2 \bar{x}_3 \vee m_6 x_1 x_2 x_3 \vee \\ & m_1 \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee m_5 \bar{x}_1 \bar{x}_2 x_3 \vee \\ & m_3 \bar{x}_1 x_2 \bar{x}_3 \vee m_7 \bar{x}_1 x_2 x_3 \\ = {} & m_1 \bar{x}_1 \bar{x}_2 \bar{x}_3 \vee m_5 \bar{x}_1 \bar{x}_2 x_3 \vee \\ & m_3 \bar{x}_1 x_2 \bar{x}_3 \vee m_7 \bar{x}_1 x_2 x_3 \vee \\ & m_0 x_1 \bar{x}_2 \bar{x}_3 \vee m_4 x_1 \bar{x}_2 x_3 \vee \\ & m_2 x_1 x_2 \bar{x}_3 \vee m_6 x_1 x_2 x_3, \end{aligned}$$

which gives $m_1 m_5 m_3 m_7 m_0 m_4 m_2 m_6$ as the binary representation of $f(x_3, x_2, \bar{x}_1)$. Several randomly chosen NP-equivalents of $f(x_1, x_2, x_3)$ are shown in Fig. 1, where the binary representations are written vertically; in the subsequent discussions, binary representations will often be displayed in this way.

3.2 Basic Idea

Figure 1 shows four of the NP-equivalents of $f(x_1, x_2, x_3)$. There are at most 48 ($= 3!2^3$) NP-equivalents of a three-variable function. Our objective in computing the NPR is to find the NP-equivalent that has the smallest value in the binary representation. Thus, we can generate NP-equivalents with other permutations and complementations of the variables, and take the function that has the smallest value in the

binary representation as the NPR.

An observation to the minterms of the first and second columns of Fig. 1 shows that all the minterms of $f(x_1, x_2, x_3)$ move to new positions in $f(x_1, \bar{x}_3, x_2)$. For example, the first minterm, m_0, of $f(x_1, x_2, x_3)$ becomes the second minterm of $f(x_1, \bar{x}_3, x_2)$. Therefore, each time we want to change the permutation and complementation of the variables of an n-variable function, we must compute the new positions for all the 2^n minterms. Since an n-variable function has at most $n!2^n$ NP-equivalents, to compute the NPR for an n-variable function we must compute $n!2^{2n}$ new positions for the minterms, i.e., the time complexity of the algorithm is $O(n!2^{2n})$. As a result the method requires significant amount of computation time even for functions with as few as three variables.

3.3 NP-Transformation Table (NPTT)

Figure 1 shows that the new positions of the minterms are fixed for each of the permutation and complementation of the variables. Therefore, our strategy is to compute the new positions of the minterms for all the permutation and complementation of the variables only once and to use them repeatedly for computing NPRs; this method is much faster than the method presented in Sect. 3.2, because repeated computation of the new positions for the minterms is unnecessary. Figure 2 shows a table of all such new positions of the minterms for three-variable function; it is similar to Fig. 1 except column headings are removed and m_i is replaced by i ($0 \leq i \leq 7$). We will refer to such a table as *NP-transformation table (NPTT)*. Although column headings are removed from Fig. 2 for ease of showing the whole table, they are required by our algorithm.

The NPTT for an n-variable function has 2^n rows and $n!2^n$ columns, i.e., it has $n!2^{2n}$ entries (Fig. 2). Table 1 shows the maximum number of NP-equivalents in an NP-equivalence class and the size of NPTTs for different number of variables. Since the size of the NPTTs grow exponentially, they can be practically used for functions with up to seven variables, which is the upper bound on the variables for which our Boolean matching technique can be applicable. It should be noted that the maximum number of inputs to the cells in many cell libraries is less than seven. For example, *lib2* library from MCNC has cells with up to six inputs [32].

$f(x_1, x_2, x_3)$	$f(x_1, \bar{x}_3, x_2)$	$f(\bar{x}_3, x_2, \bar{x}_1)$	$f(\bar{x}_2, \bar{x}_1, x_3)$
m_0	m_2	m_5	m_6
m_1	m_0	m_1	m_7
m_2	m_3	m_7	m_2
m_3	m_1	m_3	m_3
m_4	m_6	m_4	m_4
m_5	m_4	m_0	m_5
m_6	m_7	m_6	m_0
m_7	m_5	m_2	m_1

Fig. 1 Four of the NP-equivalents of a three-variable function $f(x_1, x_2, x_3)$.

3.4 Breadth-First Search by Using NPTT

The straightforward method for computing NPR by using NPTT that we presented in Sect. 3.3 first generates all the $n!2^n$ NP-equivalents from a given n-variable function, and then chooses one with the smallest value in the binary representation as the NPR. Since we are interested only in the function that has the smallest value in the binary representation, we may avoid generating many of the NP-equivalents. Each columns of the NPTT corresponds to an NP-equivalent, and we use a breadth-first search technique to early detect the columns that cannot lead to the NPR. In this method, the row at the top of the NPTT is used at first to generate the first minterms of all the NP-equivalents, where the first minterm is the left most minterm in the binary representation. After generating the first minterms corresponding to all the columns of the NPTT, we apply the following:

(a) if the minterms have both 0 and 1 values, we only keep the columns that generate minterms with only 0 value and discard other columns, and
(b) if all the minterms have either 0 or 1 values, we keep all the columns.

Since NPR has the smallest value in the binary representation among all the NP-equivalents, step (a) effectively discards some of the columns that cannot lead to the NPR. We then use second row of the NPTT for generating the second minterms correspond to the columns that we kept in steps (a) and (b), and apply steps (a) and (b) on the sec-

Table 1 Maximum number of NP-equivalents and size of NPTTs for different number of variables.

Number of variables	Maximum number of NP-equivalents	Size of NPTTs
3	48	384
4	384	6144
5	3840	122880
6	46080	2949120
7	645120	82575360
8	10321920	2.64×10^9

0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
1	1	2	2	4	4	0	0	3	3	5	5	0	0	3	3	6	6	1	1	2	2	7	7
2	4	1	4	1	2	3	5	0	5	0	3	3	6	0	6	0	3	2	7	1	7	1	2
3	5	3	6	5	6	2	4	2	7	4	7	1	4	1	7	4	7	0	5	0	6	5	6
4	2	4	1	2	1	5	3	5	0	3	0	6	3	6	0	3	0	7	2	7	1	2	1
5	3	6	3	6	5	4	2	7	2	7	4	4	1	7	1	7	4	5	0	6	0	6	5
6	6	5	5	3	3	7	7	4	4	2	2	7	7	4	4	1	1	6	6	5	5	0	0
7	7	7	7	7	7	6	6	6	6	6	6	5	5	5	5	5	5	4	4	4	4	4	4

4	4	4	4	4	4	5	5	5	5	5	5	6	6	6	6	6	6	7	7	7	7	7	7
0	0	5	5	6	6	1	1	4	4	7	7	2	2	4	4	7	7	3	3	5	5	6	6
5	6	0	6	0	5	4	7	1	7	1	4	4	7	2	7	2	4	5	6	3	6	3	5
1	2	1	7	2	7	0	3	0	6	3	6	0	3	0	5	3	5	1	2	1	4	2	4
6	5	6	0	5	0	7	4	7	1	4	1	7	4	7	2	4	2	6	5	6	3	5	3
2	1	7	1	7	2	3	0	6	0	6	3	3	0	5	0	5	3	2	1	4	1	4	2
7	7	2	2	1	1	6	6	3	3	0	0	5	5	3	3	0	0	4	4	2	2	1	1
3	3	3	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	0	0	0	0	0	0

Fig. 2 NP-transformation table (NPTT) for three variables.

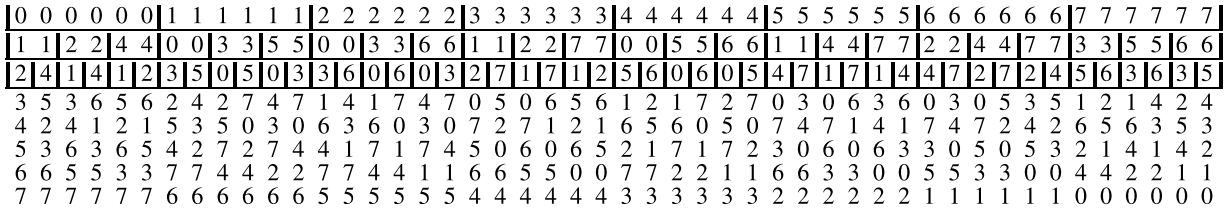

Fig. 3 Partitioning NPTT for three variables.

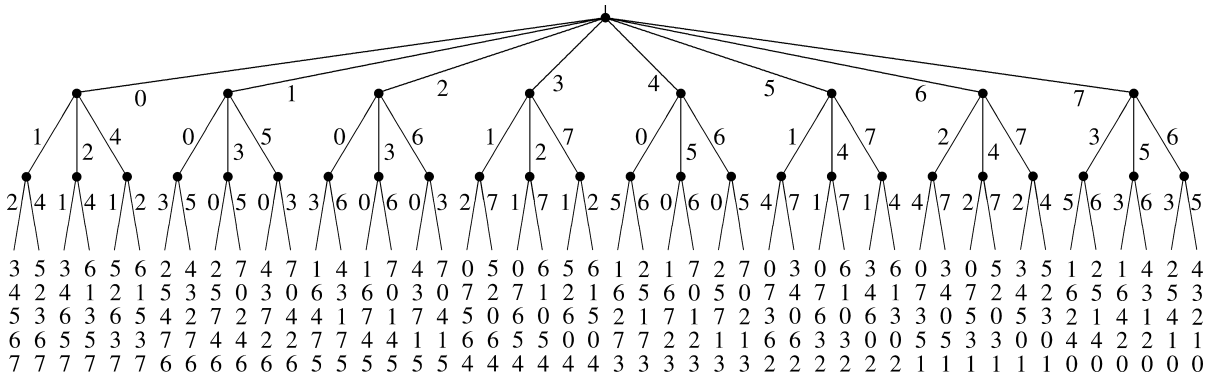

Fig. 4 Breadth-first search tree combined with partial NPTT for three variables.

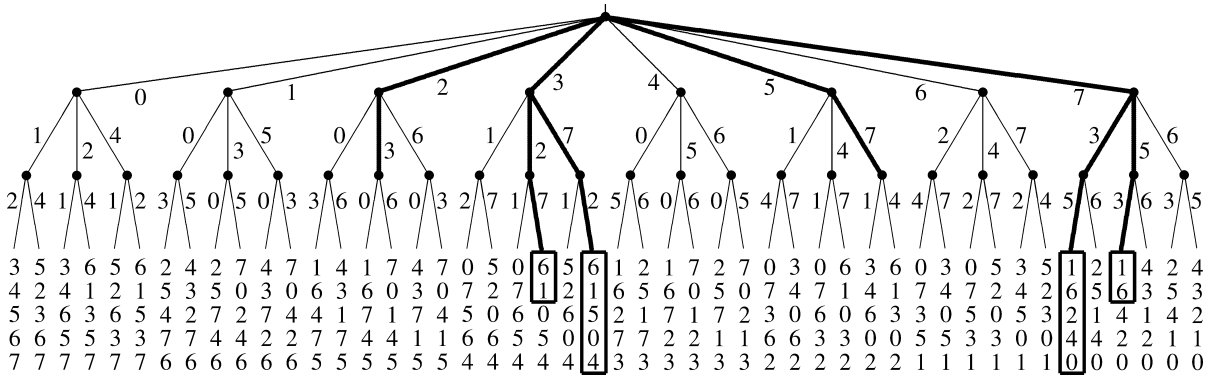

Fig. 5 Tree search for three-variable function 11001010_2.

ond minterms for possibly discarding some of the remaining columns from consideration. We continue this process until the bottom row of the NPTT is considered. At this point search terminates, and any of the remaining columns can generate an NPR. It should be noted that the breadth-first search technique is difficult to apply if we cannot store NPTT.

3.5 Combining NPTT with a Breadth-First Search Tree

Although the breadth-first search by using NPTT can reduce the search space quickly, by combining a search tree with the NPTT we can make the search even more efficient. In Sect. 3.4, NPTT is used row by row for computing NPR. The breadth-first search by using NPTT in Fig. 2 requires first to check all the 48 elements on the first row. An observation to Fig. 2 shows that there are 8 distinct elements on the first row. Therefore, we can partition these elements in to 8 groups and perform 8 checks — instead of 48 — to determine any columns that cannot lead to NPR; in this case also we use the breadth-first search strategy that are used in steps (a) and (b) of Sect. 3.4; but the number of checks here is only 8. Partitioning of NPTTs in Fig. 2 is shown in Fig. 3.

Figure 2 also shows that each of these groups can be partitioned in to 3 subgroups, which in turn can be again partitioned in to 2 subgroups (Fig. 3). These lead to a *breadth-first search tree* shown in Fig. 4, where the branches of the tree are labeled with the elements of the first three rows of the NPTT. Figure 4 shows that the top three rows of Fig. 2 form the search tree, while the bottom five rows stay the same. Therefore, the search for an NPR starts at the root of the search tree; after reaching the bottom of the tree the search continues from the fourth row of the NPTT until its bottom row is considered.

The breadth-first search tree for an n-variable function can be constructed in a similar manner. The root node of the

search tree has 2^n children; the number of children for each of the nodes in the subsequent levels has $n, n-1, \ldots, 3$, and 2 children.

Figure 5 shows an example to find the NPR for three-variable function 11001010_2, i.e., $m_2 = m_3 = m_5 = m_7 = 0$ and $m_0 = m_1 = m_4 = m_6 = 1$. At the top level of the tree, we discard branches for m_0, m_1, m_4, and m_6, because these minterms have a value 1. We are only interested to find the function with the smallest binary representation. Thus, the paths for m_2, m_3, m_5, and m_7 are selected. In the next level, in a similar manner, we select only six branches. We continue this process until we reach the leaves of the tree. The branches that we traverse to find the NPR are shown in thick lines or small rectangles in Fig. 5. The number of branches left at different levels of the tree are 4, 6, 4, 4, 4, 2, 2, 2. Only two paths lead to the NPR which is 00011011_2. In this way, we need to search only a small portion of the tree to find the NPR.

4. Experimental Results

We implemented the proposed Boolean matching technique for functions with up to seven variables on a Sun Fire 280R Server. The program requires about 140 megabytes memory, of which about 85 megabytes are used to store the NP-transformation tables (NPTTs); the data structure of the breadth-first search trees and the code of the program use the remaining 55 megabytes. If the program is used for functions with up to six variables, it needs only about five megabytes memory. It should be noted that additional memory is required to store NP-representatives (NPRs) for the library functions and their associated hash tables; however, this memory requirement is relatively lower as every byte of memory can hold up to eight bits of the binary representation of NPRs. During setup phase, the program constructs the breadth-first search trees; it takes about 0.50 seconds.

To demonstrate the effectiveness of our technique, we conducted an experiment by using 5,000,000 pseudo-random functions with three to seven variables and tried to match them against a cell library, which is represented by 50,000 randomly generated library functions. Table 2 summarizes the average Boolean matching time in microseconds, which is the time required to match a function against the entire library whose cells generate 50,000 library functions. We note that no two functions in the library are NP-equivalent, and Boolean matching time of our algorithm is almost independent of whether or not a matching is found. Our experiments show that Boolean matching time remains the same even when drastic changes are made in the composition of the library.

Table 2 Average time for Boolean matching.

Number of variables	Time (microseconds)
3	9.81
4	15.74
5	26.02
6	39.13
7	147.61

Hinsberger and Kolla reported Boolean matching time for their TEMPLATE technology mapping system in [12]. From multiple-level networks of NOR gates, TEMPLATE generates all possible single-output *cluster functions* [9] with six and fewer variables [17]. It then tries to find a Boolean match for each of the cluster functions against the *lib2* library from the MCNC. In library binding of a set of 18 benchmark functions by using *lib2* — which consists of 27 cells — TEMPLATE checks total 113,188 Boolean matchings in 9,141 seconds on an HP 735/125, i.e., on the average 12.38 matching attempts per second.

Since experimental results for both the systems are unavailable in the same format, a comparison of the speed performance of our Boolean matching technique with that of the TEMPLATE is not straightforward. We consider that a Sun Fire 280R Server (900-MHz UltraSPARC-III processor) is about eight times faster than an HP 735/125 (usually 125-MHz PA-7150 RISC processor). Thus, if all the cluster functions generated by TEMPLATE depends on four, five, and six variables, our method is about 600, 400, and 250 times, respectively, faster than TEMPLATE. The actual speed-up would be in between these numbers. It should be noted that the comparison is made when TEMPLATE uses a library with only 27 cells and our method uses a library with 50,000 cells.

5. Conclusions and Comments

Fast algorithms for Boolean matching can significantly speed-up the cell library binding process, and Boolean matching for cell library binding where the library contains large number of cells can considerably improve the quality of the solutions. We used the notion NP-representative (NPR) which is unique for any NP-equivalence classes, and presented a table look-up based breadth-first search algorithm to quickly compute it; we used NPRs to efficiently check the functional equivalence of a given circuit against a large library under permutation and complementation of inputs and complementation of output. The method is more than two orders of magnitude faster than Hinsberger-Kolla's algorithm [12].

Our technique is practical for functions with up to seven variables. This number is sufficiently large to work with many cell libraries, such as an industrial library reported by Benini et al. that contains cells with up to five inputs [3] and *lib2* library that consists of cells with up to six inputs and that is extensively used by the research community [32]. Recently, Abdollahi and Pedram presented a Boolean matching method that uses functional properties and can handle functions with more variables than our method does [1]. However, our method is independent of functional properties and has a comparable speed performance to work with many practical cell libraries. Although the memory usage of our method is relatively higher than most other Boolean matching algorithms, we believe its su-

perior speed performance and the ability to handle large libraries would outweigh any considerations for its memory requirement.

Acknowledgments

We thank Professor Reiner Kolla for interesting discussions about Boolean matching in TEMPLATE system. 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] A. Abdollahi and M. Pedram, "A new canonical form for fast Boolean matching in logic synthesis and verification," Proc. ACM/IEEE 42nd Design Automation Conf., pp.379–384, June 2005.

[2] 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.

[3] L. Benini, P. Vuillod, and G. De Micheli, "Iterative remapping for logic circuits," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.17, no.10, pp.948–964, Oct. 1998.

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

[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," Form. Methods Syst. Des., 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 by efficient computation of canonical form," IEICE Trans. Fundamentals, vol.E87-A, no.12, pp.3134–3140, Dec. 2004.

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

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

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

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

[13] S.L. Hurst, D.M. Miller, and J.C. Muzio, Spectral Techniques in Digital Logic, Academic Press, 1985.

[14] M. Hütter and M. Scheppler, "Memory efficient and fast Boolean matching for large functions using rectangle representation," ACM/IEEE Int. Workshop on Logic Synthesis, pp.164–171, May 2003.

[15] V. Kantabutra, "Two new directions in low-power digital CMOS VLSI design," in Low-Voltage/Low-Power Integrated Circuits and Systems, ed. E. Sánchez-Sinencio and A.G. Andreou, IEEE Press, 1999.

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

[17] R. Kolla, Personal communication, June 2003.

[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] C. Liem and M. Lefebvre, "Performance directed technology mapping using constructive matching," ACM/IEEE Int. Workshop on Logic Synthesis, vol.3, 1991.

[20] 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.

[21] 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.

[22] J. Mohnke and S. Malik, "Permutation and phase independent Boolean comparison," Proc. IEEE European Conf. on Design Automation, pp.86–92, Feb. 1993.

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

[24] T. Sasao, Switching Theory for Logic Synthesis, Kluwer Academic Publishers, 1999.

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

[26] U. Schlichtmann, F. Brglez, and P. Schneider, "Efficient Boolean matching based on unique variable ordering," ACM/IEEE Int. Workshop on Logic Synthesis, pp.3b:1–3b:13, May 1993.

[27] E. Schubert and W. Rosenstiel, "Combined spectral techniques for Boolean matching," Proc. ACM Int. Symposium on Field-Programmable Gate Arrays, pp.38–43, Feb. 1996.

[28] V. Tiwari, P. Ashar, and S. Malik, "Technology mapping for low power in logic synthesis," Integr. VLSI J., vol.20, no.3, pp.243–268, July 1996.

[29] 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.

[30] 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.

[31] 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.

[32] S. Yang, Logic Synthesis and Optimization Benchmarks User Guide (Version 3.0), Technical Report, Microelectronics Center of North Carolina (MCNC), Jan. 1991.

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 for papers presented at ISMVLs in 1986, 1996, 2003 and 2004, 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.