Row-Shift Decompositions for Index Generation Functions

Tsutomu Sasao

Kyushu Institute of Technology
Iizuka 820-8502, Japan

***Abstract*—This paper shows a realization of incompletely specified index generation functions in the form $f(X_1, X_2) = g(h(X_1)+X2)$, where $+$ denotes an integer addition. A decomposition algorithm is shown. Experimental results show that most of $n = 2q-3$ variable functions where $k = 2^q - 1$ combinations are specified can be realized by a pair of q-input q-output LUTs. The computation time is $O(k)$. Experimental results using address tables, lists of English words, and randomly generated functions are shown.**

***Index Terms*—Incompletely specified function, random function, linear transform, functional decomposition, data compression, IP address, hash function.**

I. Introduction

Given a logic function f of n variables, a straightforward method to implement f is to use a look-up table (LUT)[1]. However, the size of the LUT to implement f is proportional to 2^n. When n is large, a realization of f by a single LUT is, in many cases, impractical. **Functional decomposition** [1], [2] is a technique to realize a function using smaller LUTs by the circuit shown in Fig. 1.1. It realizes a function in the form: $f(X_1, X_2) = g(h(X_1), X_2)$, where $X_1 = (x_p, x_{p-1}, \ldots, x_1)$ and $X_2 = (x_n, x_{n-1}, \ldots, x_{p+1})$. Unfortunately, only a small fraction of the functions have functional decompositions. Also, no efficient algorithm is know to find the decomposition of type Fig. 1.1.

In this paper, we present a new type of decomposition, the **row-shift decomposition** shown in Fig. 1.2. It realizes an incompletely specified function in the form: $f(X_1, X_2) = g(h(X_1)+X_2)$, where $+$ denotes an integer addition. This decomposition efficiently realizes sparse incompletely specified functions.

Assume that the given n-variable function f is defined for only k input combinations, and $n = 2q - 3$, where $q = \lceil \log_2(k+1) \rceil$. This paper claims that most functions can be implemented by the circuit shown in Fig. 1.2, where the LUTs for G and H have at most q inputs and q outputs.

The rest of the paper is organized as follows: Section 2 defines index generation functions; Section 3 shows a method to reduce the number of variables for incompletely specified index generation functions; Section 4 shows the row-shift decomposition that realizes an index generation function by a pair of LUTs; Section 5 explains why the row-shift decomposition works; Section 6 shows experimental results; and finally Section 7 concludes the paper.

II. Index Generation Function

Definition 2.1: Consider a set of k different vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique index from 1 to k. A **registered vector table** shows the **index** for each registered vector. An **incompletely specified index generation function** produces a corresponding index when the input vector matches a registered vector. Otherwise, the function produces d (*don't care*). In this case, k is the **weight** of the function. The incompletely specified index generation function represents a mapping $B^n \rightarrow \{d, 1, 2, \ldots, k\}$, or $D \rightarrow \{1, 2, \ldots, k\}$, where $D \subset B^n$ denotes the set of registered vectors.

Example 2.1: Consider the registered vectors shown in Table 2.1. These vectors show an index generation function with weight $k = 7$. ■

Index generation functions are useful for address tables in the internet, terminal access controller of local area networks, database, memory patch circuits, text compression, password tables, and code converters [7]. Such circuits must be updated frequently, thus programmable or reconfigurable realizations are desirable.

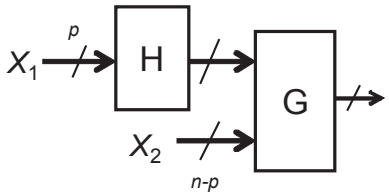

Fig. 1.1. Conventional Functional Decomposition

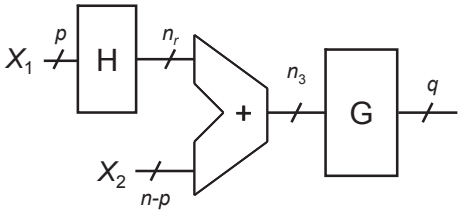

Fig. 1.2. Row-Shift Decomposition

[1]978-3-9810801-8-6/DATE12/©2012 EDAA

TABLE 2.1
REGISTERED VECTOR TABLE

Vector						Index
x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

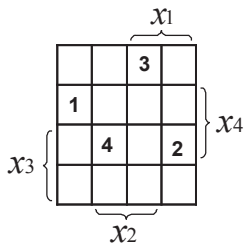

Fig. 3.1. Index Generation Function of 4 variables.

TABLE 3.1
THE AVERAGE NUMBER OF VARIABLES TO REPRESENT INDEX GENERATION FUNCTIONS OF m VARIABLES WITH WEIGHT k.

k	$m = 16$	$m = 20$	$m = 24$	n
7	3.052	3.018	3.003	3
15	4.980	4.947	4.878	5
31	6.447	6.115	6.003	7
63	8.257	8.007	8.000	9
127	10.304	10.000	9.963	11
255	12.589	11.996	11.896	13
511	14.890	14.019	13.787	15
1023	15.991	16.293	15.874	17
2047	16.000	18.758	17.965	19
4095	16.000	19.992	20.093	21

$n = 2\lceil \log_2(k+1) \rceil - 3$.

TABLE 3.2
REGISTERED VECTOR TABLE

Vector							Index
x_1	x_2	x_3	x_4	x_5	x_6	x_7	
1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	2
0	0	1	0	0	0	0	3
0	0	0	1	0	0	0	4
0	0	0	0	1	0	0	5
0	0	0	0	0	1	0	6
0	0	0	0	0	0	1	7

III. REDUCTION OF VARIABLES IN INCOMPLETELY SPECIFIED FUNCTIONS

In an incompletely specified function f, *don't care* values can be chosen as either 0 or 1 to minimize the number of variables to represent f. This property is useful to realize the function using smaller LUTs.

Lemma 3.1: Suppose that an incompletely specified function f is represented by a decomposition chart. If each column has at most one *care* element, then f can be represented by using only the column variables.

(Proof) In each column, let the values of *don't cares* elements be set to the value of the *care* element in the column, then the function depends only the column variables. □

Example 3.1: Consider the decomposition chart shown in Fig. 3.1, where x_1 and x_2 specify the columns, and x_3 and x_4 specify the rows, and blank elements denote *don't cares*. Note that in Fig. 3.1, each column has at most one *care* element. Thus, this function can be represented by only the column variables x_1 and x_2:

$$f = 1 \cdot \bar{x}_1\bar{x}_2 \vee 4 \cdot \bar{x}_1 x_2 \vee 3 \cdot x_1 x_2 \vee 2 \cdot x_1\bar{x}_2.$$

■

Algorithms to minimize the number of variables in incompletely specified functions are known [3], [4], [6]. Table 3.1 shows the average number of variables n to represent random incompletely specified index generation functions of n variables with weight k. We have the following:

Conjecture 3.1: [7] When the number of the input variables is sufficiently large, more than 95% of incompletely specified index generation functions with weight k ($\geq$ 7), can be represented with $n = 2\lceil \log_2(k+1) \rceil - 3$ variables.

For most functions, the necessary number of variables n depends on only k, and is independent of m, the number of variables in the original functions. For example, most index generation functions with weight $k = 7$ can be represented by three variables. *Exceptions* that do not satisfy Conjecture 3.1 include:

Example 3.2: The function in Table 2.1 cannot be represented with three variables. It can be represented with four variables: x_2, x_3, x_4, x_5. ■

Example 3.3: Consider the registered vectors shown in Table 3.2. It shows an index generation function with weight $k = 7$. To distinguish these seven vectors, 6 variables are necessary. Note that in Table 3.2, each column has only one 1, and six 0's. Thus, the decision tree for this function is unbalanced. Hence, the function requires six variables to distinguish the vectors. ■

For such functions, a **linear transformation** [9] is useful to reduce the number of variables.

Example 3.4: In the registered vectors in Table 3.2, apply the following linear transformation:

$$\begin{aligned} y_1 &= x_1 \oplus x_3 \oplus x_5 \oplus x_7 \\ y_2 &= x_2 \oplus x_3 \oplus x_6 \oplus x_7 \\ y_3 &= x_4 \oplus x_5 \oplus x_6 \oplus x_7 \end{aligned}$$

Then, we have the registered vector table shown in Table 3.3. In this table, only three variables are necessary to distinguish the vectors. Note that in Table 3.3, each column has three 0's, and four 1's. Thus, the decision tree for this function is more balanced than that of Table 3.2. Hence, the function requires fewer variables than that for Table 3.2. ■

The transformation in Example 3.4 has **compound degree** four, since four variables are combined with EXOR operators. As for the lower bound on the number of variables, we have the following:

TABLE 3.3
REGISTERED VECTORS TABLE AFTER LINEAR TRANSFORMATION

Vector			Index
y_3	y_2	y_1	
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

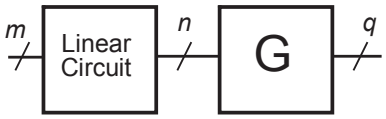

Fig. 3.2. Linear Decomposition.

Theorem 3.1: To represent any incompletely specified index generation function f with weight k, at least $q = \lceil \log_2 k \rceil$ variables are necessary.

(Proof) The number of different vectors specified with $q-1$ variables is at most $2^{q-1} < k$. Thus, at least q variables are necessary to represent an index generation function with weight k. □

Conjecture 3.1 shows that most functions with weight k can be represented with $n = 2\lceil \log_2(k+1) \rceil - 3$ variables as shown in Fig. 3.2. Although this is a considerable improvement over the straightforward implementation that uses an m-input q-output LUT, it still requires a large LUT when k is large. In the next section, we will show a more efficient realization.

IV. ROW-SHIFT DECOMPOSITION

This section introduces the **row-shift decomposition** that represents an incompletely specified index generation function by a pair of functions with fewer variables.

Example 4.1: Fig. 4.1 is the decomposition chart for the index generation function shown in Table 2.1. Since columns for $X_2 = (x_3, x_2, x_1) = (0,0,0)$, $(0,1,0)$ and $(1,0,0)$ have two *care* elements, the function cannot be represented by only the column variables $X_2 = (x_3, x_2, x_1)$. Next, consider the decomposition chart shown in Fig. 4.2 that is obtained from Fig. 4.1 by shifting one bit to the right in the rows for $X_1 = (x_6, x_5, x_4) = (0,1,1)$, $(1,0,0)$, and $(1,1,1)$. In Fig. 4.2, each column has at most one *care* element. Thus, the modified function can be represented by only the column variables $X_2 = (x_3, x_2, x_1)$. ■

TABLE 4.1
TABLES FOR h AND g

X_1				X_2			
x_6	x_5	x_4	h	x_3	x_2	x_1	g
0	0	0	0	0	0	0	1
0	0	1	0	0	0	1	5
0	1	0	0	0	1	0	2
0	1	1	1	0	1	1	7
1	0	0	1	1	0	0	3
1	0	1	0	1	0	1	4
1	1	0	0	1	1	0	–
1	1	1	1	1	1	1	6

	000	001	010	011	100	101	110	111	
000									x3
001									x2
010	1		2		3				x1
011					4				
100	5								
101									
110								6	
111			7						

x6x5x4

Fig. 4.1. Original Decomposition Chart.

Let X_1 be the row variables, and X_2 be the column variables. In Fig. 1.2, assume that the LUT for H stores the number of bits to shift (**displacement**) for each row specified by X_1, while the LUT for G stores the non-zero (*care*) value of the column after the shift operation: $h(X_1) + X_2$. Then, Fig. 1.2 realizes a given function f in the form $f(X_1, X_2) = g(h(X_1) + X_2)$.

Example 4.2: The function $f(X_1, X_2)$ in Example 4.1 can be decomposed into $g(h(X_1)+X_2)$, where h and g are shown in Table 4.1, and + denotes an unsigned integer addition of three bits. h is implemented by a 3-input 1-output LUT, while g is implemented by a 3-input 3-output LUT. Note that a straightforward realization of Table 2.1 requires a 6-input 3-output LUT. ■

	000	001	010	011	100	101	110	111	
000									x3
001									x2
010	1		2		3				x1
011					→	4			
100	→	5							
101									
110								6	
111			→	7					

x6x5x4

Fig. 4.2. Decomposition Chart After Row-Shift.

In Example 4.1, we can represent the function without increasing the columns. However, in general, we must increase the columns to represent the function. Since each column has at most one *care* element after the shift operations, at least k columns are necessary to represent a function with weight k.

In Fig. 4.1, many ways exist to shift the rows to satisfy the constraint. For example, the row for $X_1 = (x_6, x_5, x_4) = (0, 1, 0)$ can be right-shifted by one bit instead of shifting other rows. To find an optimal solution is time-consuming. We use the **first-fit method** [10], which is simple and efficient.

Algorithm 4.1: (Find row displacements)

1) Sort the rows in decreasing order by the number of *care* elements they contain.
2) Compute the row displacement for each row at a time, where the row displacement $r(i)$ for row i has the smallest value such that no *care* element in row i is in the same position as any *care* element in the *previous* rows.

When the distribution of *care* elements among the rows is uniform, Algorithm 4.1 compresses the table effectively. To reduce the total size of LUTs, we use the following:

Algorithm 4.2: (Row-shift Decomposition)

1) Reduce the number of the variables by the method [3], [4], [7]. If necessary, use a linear transformation to further reduce the number of the variables. Let n be the number of variables after reduction.
2) Let $q_1 = \lceil \frac{n}{2} \rceil$. For $t = -2$ to $t = 2$ perform Steps 3 through 5.
3) Partition the inputs X into (X_1, X_2) [2] , where $X_1 = (x_p, x_{p-1}, \ldots, x_1)$ denotes the rows, $X_2 = (x_n, x_{n-1}, \ldots, x_{p+1})$ denotes the columns, and $p = q_1 + t$.
4) Obtain the row displacements by Algorithm 4.1.
5) Obtain the maximum value of the displacements, and compute the total sizes for LUTs.
6) Find a t that minimizes the total size of LUTs.

In Step 2, when $p = \lceil \frac{n}{2} \rceil$, the total memory size $n_r 2^p + q 2^{n_3}$ takes its minimum in Fig. 1.2 when the amount of displacements are small. However, for some functions, the total memory sizes take their minimum when $p = \lceil \frac{n}{2} \rceil - t$, where $t = -2, -1, 1$, or 2.

V. Analysis of the Method

In this part, we consider why Algorithm 4.2 obtains good solutions for most functions. By Conjecture 3.1, we can reduce the number of variables to satisfy the following relation:

$$n = 2\lceil \log_2(k+1) \rceil - 3.$$

Property 5.1: Most uniformly distributed incompletely specified index generation functions of n variables with weight k can be realized by a pair of q-input q-output LUTs, when $q = \lceil \log_2(k+1) \rceil$ and $n = 2q - 3$.

[2]Unlike ordinary functional decompositions, the influence of the partition (X_1, X_2) is relatively small. In the row-shift decomposition, we can assume that *care* elements are uniformly distributed in the decomposition chart.

(Explanation Supporting the Property) The probability that a function f takes a *care* (non-zero) value is $\alpha = \frac{k}{2^n}$. The probability that a function f takes a *don't care* value is $\beta = 1.0 - \alpha$. Consider the decomposition chart, where the number of the row variables is n_1, the number of the column variables is n_2, and $n_1 + n_2 = n$. Let $N_1 = 2^{n_1}$ and $N_2 = 2^{n_2}$. The probability that a row has all don't care elements is

$$a_0 = \beta^{N_2}.$$

The probability that a row has t (≥ 1) *care* elements is

$$a_t = \binom{N_2}{t} \alpha^t \beta^{N_2 - t}.$$

When α is sufficiently small, $\beta = 1.0 - \alpha$ can be approximated by $e^{-\alpha}$ [7]. Thus, we have

$$a_t \simeq \frac{N_2^t}{t!} \alpha^t e^{-\alpha(N_2 - t)} = \frac{e^{-\alpha N_2}}{t!} (N_2 \alpha e^{\alpha})^t$$

Assume that $k + 1 = 2^{n_1} = N_1$. Since $\alpha \simeq \frac{2^{n_1}}{2^n} = \frac{1}{2^{n_2}} = \frac{1}{N_2}$, we have

$$a_t \simeq \frac{e^{-1}}{t!} e^{\alpha t}.$$

Since $\alpha \simeq \frac{1}{N_2}$, and N_2 is sufficiently large, αt is sufficiently small. Thus, $e^{\alpha t}$ is approximated by 1.0. In this case, we have:

$$\begin{aligned}
a_0 &\simeq \beta^{N_2} \simeq e^{-\alpha N_2} \simeq e^{-1} \simeq 0.3678. \\
a_1 &\simeq \frac{e^{-1}}{1!} e^{\alpha} \simeq e^{-1} \simeq 0.3768. \\
a_2 &\simeq \frac{e^{-1}}{2!} e^{2\alpha} \simeq \frac{e^{-1}}{2} \simeq 0.1839. \\
a_3 &\simeq \frac{e^{-1}}{3!} e^{3\alpha} \simeq \frac{e^{-1}}{6} \simeq 0.0613. \\
a_4 &\simeq \frac{e^{-1}}{4!} e^{4\alpha} \simeq \frac{e^{-1}}{24} \simeq 0.0153. \\
a_5 &\simeq \frac{e^{-1}}{5!} e^{5\alpha} \simeq \frac{e^{-1}}{120} \simeq 0.0031.
\end{aligned}$$

In other words, 36.8% of the rows have all *don't care* elements; 37.7% of the rows have just one *care* element; 18.4% of the rows have two *care* elements; 6.13% of the rows have three *care* elements; 1.5% of the rows have four *care* elements; and 0.31% of the rows have five *care* elements.

Algorithm 4.1 modifies the decomposition chart so that each column has at most one *care* element. In this case, the number of columns after the row shift operations is at least k. From the hypothesis, we have $n = 2q - 3$ and $n_1 = q$. Thus, we have $n_2 = q - 3$ and $N_2 = 2^{n_2} = 2^{q-3} \simeq \frac{k}{8}$.

In other words, the number of columns of the decomposition chart is approximately $\frac{1}{8}$ of the number of rows. Thus, after the row-shift operations, the number of columns is increased to 8 times of the original size. In the circuit realization, the LUT for h has q inputs and $\lceil \log_2(8 \cdot 2^{n_2}) \rceil = 3 + n_2 = q$ outputs, also the LUT for g has q inputs and q outputs.

Experimental results show that, in many cases, $k + 1$ columns are sufficient to represent the functions. This is interpreted as follows: In Algorithm 4.1, the positions of rows are determined in the decreasing order of the number of *care*

TABLE 6.1
REALIZATIONS OF IP ADDRESS TABLE (WITHOUT LINEAR TRANSFORMATIONS. $m = 32$).

k	n	n_1	n_2	n_r	n_3	q	Total bits	$q2^n$
1670	18	9	9	11	11	11	28160	2.88×10^6
3288	20	10	10	12	12	12	61440	1.26×10^7
4591	21	9	12	13	13	13	113152	2.73×10^7
7903	23	12	11	13	13	13	159744	1.09×10^8

elements. Thus, a row with the most *care* elements is not shifted. Other rows may be shifted to the right so that each column has at most one *care* element. Note that, the rows with only one *care* element can be shifted to right positions to fill any *unused* columns. This is the reason why the displacements of rows are determined in the decreasing order of the *care* elements.

In the original decomposition chart, if the leftmost two columns have all *don't cares*, then these columns cannot be filled by Algorithm 4.2, since only the shift right operations are permitted. In such a case, Property 5.1 does not hold. However, the probability of such case is very small:

$$\beta^{2N_1} \simeq e^{-2\alpha N_1} = e^{-\frac{2N_1}{N_2}} = e^{-16} = 1.125 \times 10^{-7}.$$

From this, we have the property. □

In the next section, we verify this by experiments.

VI. EXPERIMENTAL RESULTS

We implemented Algorithm 4.2, and applied to three different classes of index generation functions.

A. IP Address Tables

As for the data, we used distinct IP addresses of computers that accessed our web site over a period of a month. We considered four lists of different values of k. Table 6.1 shows the results. Note that the original number of variables is $m = 32$. The first column shows k, the number of registered vectors. The second column shows n, the number of variables after reduction of the variable using the method [6]. The third column shows n_1, the number of variables in X_1. The fourth column shows n_2, the number of variables in X_2. The fifth column shows n_r, the number of output bits for LUT H. The sixth column shows n_3, the number of variables for LUT G. The seventh column shows q, the number of output bits for LUT G, which is equal to $\lceil \log_2(k+1) \rceil$. The eighth column shows $n_r 2^{n_1} + q2^{n_3}$, the total memory size to implement the function in Fig. 6.1. The last column shows $q2^n$, the memory size to implement the function by the implementation in Fig. 3.2 . As shown in Table 6.1, in all cases, the total sizes of LUTs are smaller than the sizes of the LUTs in Fig. 3.2.

B. List of English Words

To compress English text, a list of frequently used words is useful [5]. We made three lists of English words: *List A*, *List B*, and *List C*. The maximum number of characters in the word lists is 13, but we only consider the first 8 characters. For English words consisting of fewer than 8 letters, we append blanks to make the length of words 8. We represent each

TABLE 6.2
REALIZATIONS OF ENGLISH WORDS LISTS (WITHOUT LINEAR TRANSFORMATIONS. $m = 40$).

k	n	n_1	n_2	n_r	n_3	q	Total bits	$q2^n$
1730	31	16	15	7	15	11	819200	2.36×10^{10}
3366	31	16	15	7	15	12	851968	2.57×10^{10}
4705	37	19	18	8	19	13	7602176	1.78×10^{12}

TABLE 6.3
REALIZATIONS OF ENGLISH WORDS LISTS (WITH LINEAR TRANSFORMATIONS. $m = 40$).

k	n	n_1	n_2	n_r	n_3	q	Total bits	$q2^n$
1730	19	10	9	11	11	11	33792	5.77×10^6
3366	21	11	10	12	12	12	73728	2.52×10^7
4705	20	12	12	13	13	13	159744	2.73×10^7

alphabetic character by 5 bits. So, in the lists, all the words are represented by $m = 40$ bits. The numbers of words in the lists are 1730, 3366, and 4705, respectively. Within each word list, each English word has a unique index, an integer from 1 to k, where $k = 1730$ or 3360 or 4705. Table 6.2 shows the row-shift realization of the lists. The symbols denote the same things as Table 6.1. These functions require more variables than the values given by Conjecture 3.1. For example, consider the case of $k = 1730$. Table 6.2 shows that the function requires $n = 31$ variables after the minimization of variables by the method [6]. However, Conjecture 3.1 shows that, to realize uniformly distributed functions, $2\lceil \log_2(k+1) \rceil - 3 = 2 \times 11 - 3 = 19$ variables are sufficient. In the case of English word lists, the distributions of vectors are not uniform. In this case, Property 5.1 does not hold. To make the distribution of *care* elements uniform, we used linear transformations of compound degree two [9]. Fig. 6.1 shows the circuit. Table 6.3 shows the results when the

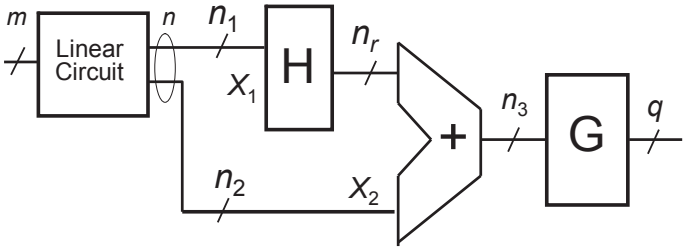

Fig. 6.1. Row-shift Decomposition with Linear Transform.

input variables are reduced by a linear transformation. Note that more variables were reduced by linear transformations than the simple reduction of variables (Table 6.2). In this case, Property 5.1 holds.

C. Random Index Generation Functions

To assess the influence of the skew of the distribution in IP address tables and English word lists, we produced random index generation functions with the same value of m and k. Table 6.4 shows the results, where the upper four rows correspond to IP address tables, while the lower three rows

TABLE 6.4
REALIZATIONS OF RANDOM FUNCTIONS.

k	n	n_1	n_2	n_r	n_3	q	Total bits	$q2^n$
1670	18	9	9	11	11	11	28160	2.88×10^6
3288	20	10	10	12	12	12	61440	1.26×10^7
4591	21	9	12	13	13	13	113152	2.73×10^7
7903	23	12	11	13	13	13	159744	1.09×10^8
1730	19	10	9	11	11	11	33792	5.77×10^6
3366	21	11	10	12	12	12	73728	2.52×10^7
4705	20	9	11	13	13	13	113152	2.73×10^7

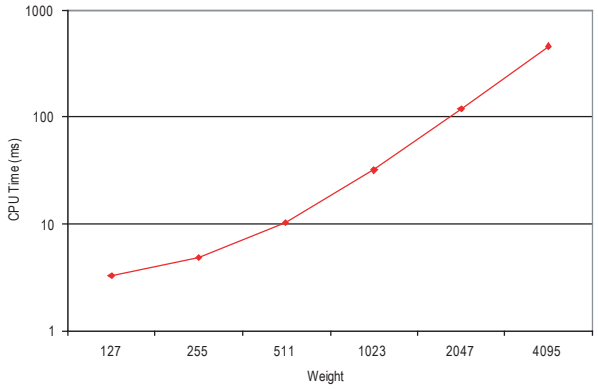

Fig. 6.2. Computation Time for Row-shift Decomposition.

correspond to English word lists. Note that the first four rows are exactly the same as Table 6.1. This shows that the IP address table is random enough when the input variables are minimized without using a linear transform.

Also, for the fifth and the sixth rows, they are exactly the same as the top two rows of Table 6.3. However, as for the last row, the random function requires a smaller LUTs. This shows that the distribution of the last English word list is not uniform enough even if the linear transformation with compound degree two is used.

D. Computation Time

The computation time is much shorter than that for conventional decomposition algorithms. Fig. 6.2 shows the average computation time (mili-seconds) for index generation functions with weight $k = 2^q - 1$ and $n = 2q - 3$ variables, where $q = 7, 8, 9, 10, 11$ and 12. Fig. 6.2 shows that the CPU time is $O(k)$. We produced 1000 index generation functions for each k. To obtain CPU time, we realized 6000 functions. Out of 6000 functions, only two required larger sizes than the sizes given by Property 5.1, and other functions satisfied Property 5.1. As for the two exceptional functions, the distribution of *care* elements are skewed. However, by using linear transformations, we can easily smooth out such skewed distributions [8]. In the experiment, we used a PC with INTEL Core 2 Duo CPU, 2.53 GHz and 3.40 GB RAM, on Windows XP Professional Operating System.

VII. CONCLUDING REMARKS

In this paper, we presented the row-shift decompositions of incompletely specified index generation functions. We also presented a heuristic algorithm to find a row-shift decompositions that reduces the total size of LUTs. When the weight of the function is much smaller than 2^n, the functions can be often represented with LUTs with fewer inputs than n.

In the previous method [7], to represent an incompletely specified index generation function with weight k, we need an LUT with $n = 2q - 3$ inputs and q outputs, where $q = \lceil \log_2(k+1) \rceil$ as shown in Fig. 3.2.

In this paper, we presented a row-shift decomposition and verified that, in many cases, the same function can be represented by a pair of LUTs with q-input and q-output. Since the previous method (Fig. 3.2) requires $q2^{2q-3}$ bits, while the presented method (Fig. 6.1) requires $(2q)2^q$ bits, the reduction ratio is $2^{q-4} \simeq \frac{k}{16}$. Thus, the presented method is effective when $k \geq 16$. Although we need an adder, the area for the adder is much smaller than the area for the LUTs reduced by the decomposition, especially when the circuit is implemented by an FPGA. Since the function can be personalized only by the LUTs, a quick update is possible. In this case, the layout and interconnections are fixed. Also, when the circuit is implemented by an FPGA, the delay of the adder can be hidden by the use of pipeline architecture.

The presented method is useful for index generation functions whose weights are small. Unfortunately, MCNC benchmark functions do not include such functions. Index generation functions are related to short-life data such as address tables, password lists, which require frequently updates. Such functions are often implemented by software. However, hardware/ firmware implementations will make them much faster.

ACKNOWLEDGMENTS

This research is partly supported by the MEXT Regional Innovation Cluster Program (Global Type, 2nd Stage), and by a Grant in Aid for Scientific Research of the JSPS. The author thanks Prof. Jon T. Butler for discussion and Mr. M. Matsuura for experiments. Comments of reviewers were useful to improve the quality of the paper.

REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions," *International Symposium on the Theory of Switching*,pp. 74-116, April 1957.
[2] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Co., Princeton, NJ, 1962.
[3] C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a Boolean function," *IEEE Transactions on Computers*, Vol. C-27, No. 11, pp. 1064-1068, November, 1978.
[4] Y. Kambayashi, "Logic design of programmable logic arrays," *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 609-617, September 1979.
[5] D. Salomon, G. Motta, and D. Bryant, *Handbook of Data Compression* (5th edition), Springer, 2009.
[6] T. Sasao, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, CA, USA, Nov.10-13, 2008, pp. 45-51.
[7] T. Sasao, *Memory-Based Logic Synthesis*, Springer, 2011.
[8] T. Sasao, "Index generation functions: Recent developments,"(invited paper) *International Symposium on Multiple-Valued Logic* (ISMVL-2011), Tuusula, Finland, May 23-25, 2011.
[9] T. Sasao, "Linear decomposition of index generation functions," *17th Asia and South Pacific Design Automation Conference* (ASPDAC-2012), Jan. 30- Feb. 2, 2012, Sydney, Australia (to appear).
[10] R. E. Tarjan, and A. C-C. Yao, "Storing a sparse table,", *Communications of the ACM*, Vol.22, No.11, Nov. 1979, pp.606-611.