

On the Numbers of Variables to Represent Sparse Logic Functions

Tsutomu Sasao

Department of Computer Science and Electronics,
Kyushu Institute of Technology,
Iizuka 820-8502, Japan

Abstract—In an incompletely specified function f , *don't care* values can be chosen to minimize the number of variables to represent f . It is shown that, in incompletely specified functions with k 0's and k 1's, the probability that f can be represented with only $p = 2^{\lceil \log_2(k+1) \rceil}$ variables is greater than $e^{-1} = 0.36788$. In the case of multiple-output functions, where only the outputs for k input combinations are specified, most functions can be represented with at most $p = 2^{\lceil \log_2(k+1) \rceil} - 1$ variables. Experimental data is shown to support this. Because of this property, an IP address table can be realized with a small amount of memory.

I. INTRODUCTION

For completely specified logic functions, logic minimization is a process of reducing the number of products to represent the given function. However, for incompletely specified functions (i.e., functions with *don't cares*), at least two problems exist [6]: The first is to reduce the number of the products to represent the function, and the second is to reduce the number of variables. The first problem is useful for sum-of-products expression (SOP)-based realizations [2], while the second problem is useful for memory-based or LUT (look-up table)-based realizations.

Example 1: Consider the four-variable function shown in Fig. 1, where the blank cells denote *don't cares*. The SOP with the minimum number of products is $\mathcal{F}_1 = x_1x_4 \vee x_2\bar{x}_3$, while the SOP with the minimum number of variables is $\mathcal{F}_2 = x_1x_2 \vee x_1x_4 \vee x_2x_4$. Note that \mathcal{F}_1 has two products and depends on four variables, while \mathcal{F}_2 has three products and depends on only three variables. x_3 is a **non-essential** variable, since \mathcal{F}_2 does not include it. (End of Example)

As shown in this example, the minimization of the number of products is different from the minimization of the number of variables. In this paper, we consider the minimization of

the number of variables. Especially, we are interested in the number of variables to represent logic functions whose values are specified for k combinations, where k is small.

The rest of the paper is organized as follows: Section II gives definitions and basic properties. Section III formulates the minimization problem of the variables in the single-output incompletely specified functions. Also, it shows that $p = 2^{\lceil \log_2(k+1) \rceil}$ variables are sufficient to represent most incompletely specified single-output functions, where k combinations are mapped into 0, k combinations mapped to 1, and the other $2^n - 2k$ combinations are mapped to *don't cares*. Section IV extends the theory to the multiple-output case. First, it introduces index generation functions, and derives the number of variables to represent an index generation function with k specified vectors. Then, it shows that $2^{\lceil \log_2(k+1) \rceil} - 1$ variables are sufficient to represent such a function in most cases. And, finally, it extends to general multiple-output functions. Section V shows an algorithm to minimize the number of variables to represent index generation functions. Section VI shows a method to convert a completely specified index generation function into an incompletely specified one. With this method, we can drastically reduce the size of memories. Section VII shows a method to reduce the number of variables by a linear transformation of the input variables. Section VIII shows experimental results for randomly generated functions, IP address tables, and multiple-output PLA benchmark functions. Finally, Section IX concludes the paper.

II. DEFINITIONS AND BASIC PROPERTIES

Definition 1: An **incompletely specified logic function** f is a mapping $D \rightarrow B$, where $D \subset B^n$, $B = \{0, 1\}$.

Definition 2: An incompletely specified logic function is represented by a pair of **characteristic functions** F_0 and F_1 , where $F_0(\mathbf{a}) = 1$ iff $f(\mathbf{a}) = 0$, and $F_1(\mathbf{a}) = 1$ iff $f(\mathbf{a}) = 1$. Note that $F_0F_1 = 0$. If $\mathbf{a} \in B^n - D$, then the value of $f(\mathbf{a})$ is unspecified, and is denoted by d (*don't care*).

Definition 3: Variables are represented by x_i ($i = 1, 2, \dots, n$). A **literal** of a variable x_i is either x_i , \bar{x}_i or the constant 1. An AND of literals is a **product term**, and an OR of products is a **sum-of-products expression** (SOP).

Example 2: Consider the function in Fig. 1. In this case $n = 4$. Table I also shows this function. The characteristic

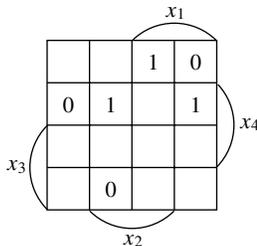


Fig. 1. Four-variable incompletely specified logic function.

TABLE I
FUNCTION FOR FIG. 1.

x_1	x_2	x_3	x_4	f
0	0	0	1	0
0	1	1	0	0
1	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	0	1

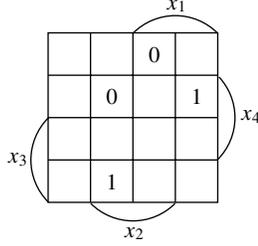


Fig. 2. Four-variable function without essential variables.

functions are

$$\begin{aligned} F_0 &= \bar{x}_1\bar{x}_2\bar{x}_3x_4 \vee \bar{x}_1x_2x_3\bar{x}_4 \vee x_1\bar{x}_2\bar{x}_3\bar{x}_4 \text{ and} \\ F_1 &= \bar{x}_1x_2\bar{x}_3x_4 \vee x_1\bar{x}_2\bar{x}_3x_4 \vee x_1x_2\bar{x}_3\bar{x}_4. \end{aligned}$$

In this case, the function is specified for only 6 minterms. (End of Example)

Definition 4: f **depends on** x_i if there exists a pair of vectors

$$\begin{aligned} \mathbf{a} &= (a_1, a_2, \dots, a_i, \dots, a_n) \text{ and} \\ \mathbf{b} &= (a_1, a_2, \dots, b_i, \dots, a_n), \end{aligned}$$

such that both $f(\mathbf{a})$ and $f(\mathbf{b})$ are specified, and $f(\mathbf{a}) \neq f(\mathbf{b})$. If f depends on x_i , then x_i is **essential** in f , and x_i must appear in every expression for f .

Definition 5: Two functions f and g are **compatible** when the following condition holds: For any $\mathbf{a} \in B^n$, if both $f(\mathbf{a})$ and $g(\mathbf{a})$ are specified, then $f(\mathbf{a}) = g(\mathbf{a})$.

Lemma 1: Let $f_0 = f(|x_i = 0)$ and $f_1 = f(|x_i = 1)$. Then, x_i is **non-essential** in f iff f_0 and f_1 are compatible. If x_i is non-essential in f , then f can be represented by an expression without x_i .

Example 3: Consider the function f in Fig. 2. It is easy to verify that all the variables are non-essential. Note that f can be represented as $\mathcal{F}_1 = \bar{x}_2 \vee x_3$ or $\mathcal{F}_2 = x_1 \oplus \bar{x}_4$. (End of Example)

Essential variables must appear in every expression for f , while non-essential variables may appear in some expressions and not in others. Algorithms to represent a given function by using the minimum number of variables have been considered [1], [4], [5], [6].

III. ANALYSIS FOR SINGLE-OUTPUT LOGIC FUNCTIONS

In this section, we derive the number of variables to represent single-output incompletely specified logic functions. In the analysis that follows, we consider a set of functions (e.g.,

all incompletely specified functions) restricted by conditions (e.g. the number of *care* values is $2k$).

Definition 6: A set of functions is **uniformly distributed**, if the probability of occurrence of any function is the same as any other function.

For example, the set of 4-variable incompletely specified functions with 1 *care* value consists of 32 members, 16 having a single 1 and 16 having a single 0. If the functions are uniformly distributed, the probability of the occurrence of any one of them is $\frac{1}{32}$.

Theorem 1: Consider a set of uniformly distributed incompletely specified function, where k combinations are mapped to 0, k combinations mapped to 1, and the other $2^n - 2k$ combinations are mapped to *don't cares*. Then, the probability that $f(x_1, x_2, \dots, x_n)$ can be represented by using only x_1, x_2, \dots, x_{p-1} , and x_p , where $p = 2\lceil \log_2(k+1) \rceil$, is greater than $e^{-1} = 0.36788$.

Proof: Let $f(X_1, X_2)$ be an incompletely specified function, where $X_1 = (x_1, x_2, \dots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \dots, x_n)$. Consider the decomposition table of $f(X_1, X_2)$, where X_1 labels the columns, and X_2 labels the rows. If no column has both 0 and 1, a completely specified function can be formed by setting all column entries to the same value, yielding a function independent of X_2 . From here, we obtain the probability PR .

Step 1: Assume that k 0's are already distributed to the decomposition table. Thus, at most k columns have 0's. Next, we distribute k 1's to the decomposition table. The probability of distributing a single 1 to a column not containing 0's is at least $\frac{2^p - k}{2^p} = 1 - \alpha$. Thus, the probability of distributing k 1's to the columns without 0's is larger than or equal to $(1 - \alpha)^k$. Hence, we have the relation:

$$PR \geq (1 - \alpha)^k. \quad (1)$$

Step 2: Next, we show that $(1 - \alpha)^k > e^{-1}$. When $0 < \alpha < 1$, $1 - \alpha$ is approximated by $e^{-\alpha}$. Thus, $(1 - \alpha)^k \approx e^{-\alpha k}$. When $p = 2\lceil \log_2(k+1) \rceil$, we have $2^p \geq (k+1)^2 > k^2$. Thus, we have $\frac{k^2}{2^p} < 1$. Therefore, $e^{-\alpha k} = e^{-\frac{k^2}{2^p}} > e^{-1}$, and

$$(1 - \alpha)^k > e^{-1}. \quad (2)$$

From (1) and (2), we have the theorem. ■

Theorem 1 shows a typical property of the randomly generated functions. Note that there exist functions that require more than $p = 2\lceil \log_2(k+1) \rceil$ variables, as shown below. However, we conjecture that the fraction of such functions is very small.

Example 4: Consider the n -variable function $f(X)$ such that $f(a_1, a_2, \dots, a_n) = 0$, when $\sum_{i=1}^n a_i = 0$, $f(a_1, a_2, \dots, a_n) = 1$, when $\sum_{i=1}^n a_i = 1$, and $f(a_1, a_2, \dots, a_n) = d$, when $\sum_{i=1}^n a_i \geq 2$. In this case, all the variables are essential. (End of Example)

In Theorem 1, variables are selected in a natural order without considering the properties of functions, i.e., $X_1 = (x_1, x_2, \dots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \dots, x_n)$. However, in practice, we can select a minimum set of variables to

TABLE II
REGISTERED VECTOR TABLE.

Vector	Index
1001	1
1111	2
0101	3
1100	4

represent the function. So, the number of variables to represent the function can be less than $2\lceil\log_2(k+1)\rceil$. From the experimental results in Section VIII, we have the following:

Conjecture 1: Consider a set of uniformly distributed functions of n variables, where k combinations are mapped to 0, k combinations are mapped to 1, and the other $2^n - 2k$ combinations are mapped to *don't cares*. Then, most functions can be represented with at most $p = 2\lceil\log_2(k+1)\rceil - 2$ variables.

IV. EXTENSION TO MULTIPLE-OUTPUT FUNCTIONS

In practical applications, many functions have multiple outputs, and the outputs values are different for different inputs. So, we now consider such a class of functions. First, we consider the class of index generation functions, which are special case of multiple-output functions. Then, we extend the theory into general multiple-output functions.

A. Index Generation Functions

Index generation functions¹ are used for IP address lists for Internet [12], memory patch circuits, password lists, etc [8].

Definition 7: Consider a set of k different binary vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique integer from 1 to k . A **registered vector table** shows the **index** of each registered vector. An **index generation function** produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. In this paper, we assume that k is much smaller than 2^n , the total number of input combinations.

Example 5: Table II shows a registered vector table consisting of 4 vectors. The corresponding index generation function produces the index represented by a 3-bit number (e.g., 001) of a matched vector. When no entry matches the input vector, the function produces 000. (End of Example)

B. Number of Variables to Represent Index Generation Functions

In this section, we derive the number of variables to represent an incompletely specified index generation function with k registered vectors. The basic idea is as follows: a function $f(X_1, X_2)$ is represented by a decomposition table, where X_1 labels the columns and X_2 labels the rows. If each column has at most one *care* element, then the function can be represented by using only variables in X_1 . The next example illustrates this.

¹Index generation functions were called **address generation functions** in previous publications [8], [9], [10].

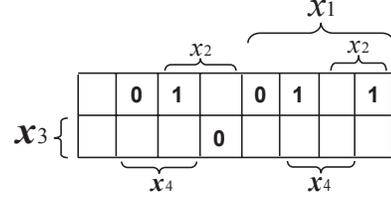


Fig. 3. Decomposition table of a four-variable function.

Example 6: Consider the decomposition table shown in Fig. 3. It shows exactly the same function as Fig. 1, but it has a different labeling of variables. In Fig. 1, x_1 and x_2 specify the columns, while in Fig. 3, x_1 , x_2 , and x_4 specify the columns. Note that in Fig. 3, each column has at most one *care* element. Thus, the function can be represented by only the variable for columns: $x_1x_2 \vee x_1x_4 \vee x_2x_4$. (End of Example) From here, we obtain the probability of such a condition by a statistical analysis.

Theorem 2: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \dots, x_n)$ with weight k , where $2 \leq k < 2^{n-2}$. When $k \leq 30000$, the probability that f can be represented with x_1, x_2, \dots , and x_p , where $p = 2\lceil\log_2(k+1)\rceil - 1$, is greater than 0.3628.

Proof: Let (X_1, X_2) be a partition of the input variables X , where $X_1 = (x_1, x_2, \dots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \dots, x_n)$. Consider the decomposition table for $f(X_1, X_2)$, where X_1 labels the column variables and X_2 labels the row variables. If each column has at most one *care* element, then f can be represented by using only X_1 . Assume that k *care* elements are distributed in the decomposition table. Then, the probability that each column has at most one *care* element is

$$\begin{aligned}
 PR &= \frac{2^p}{2^p} \cdot \frac{2^p - 1}{2^p} \cdot \frac{2^p - 2}{2^p} \cdots \frac{2^p - (k-1)}{2^p} \\
 &= 1 \cdot \left(1 - \frac{1}{2^p}\right) \cdot \left(1 - \frac{2}{2^p}\right) \cdots \left(1 - \frac{k-1}{2^p}\right) \\
 &= \prod_{i=0}^{k-1} \left(1 - \frac{i}{2^p}\right).
 \end{aligned}$$

That is, in such a distribution, ‘1’ can be placed in any column, ‘2’ can be placed in any column except that for ‘1’, etc. By exhaustive examination of the numerical values of PR for $1 \leq k \leq 30000$, we have the lemma. ■

The above theorem shows the case when the input variables are removed without considering the property of the function. In practice, we can remove the maximum number of non-essential variables by an optimization program. Thus, in many cases, the number of necessary variables is smaller than $2\lceil\log_2(k+1)\rceil - 1$. From the experimental results in Section VIII, we have the following:

Conjecture 2: Consider a set of uniformly distributed index generation functions with weight k . In most cases, an index generation function can be represented by at most $p = 2\lceil\log_2(k+1)\rceil - 1$ variables.

C. Number of Variables to Represent General Multiple-Output Functions

Theorem 3: Let F be an arbitrary n input m output function, and let D be a set of k randomly selected vectors in B^n . Let \hat{F} be the incompletely specified functions defined on only D . When $k \leq 30000$, the probability that \hat{F} can be represented with x_1, x_2, \dots , and x_p , where $p = 2\lceil \log_2(k+1) \rceil - 1$, is greater than 0.3628.

Proof: For each vector in D , assign a unique index $(1, 2, \dots, k)$. From D , we can define an incompletely specified index generation function: $D \rightarrow \{0, 1, 2, \dots, k\}$. Next, for each vector in D , obtain the output value of F , and make a truth table showing the function $\{1, 2, \dots, k\} \rightarrow B^m$. Note that this function can be implemented by memory with $\lceil \log_2(k+1) \rceil$ inputs. Thus, the incompletely specified function \hat{F} can be realized as the cascade connection of the index generation circuit and the memory.

By Theorem 2, the index generation function can be represented with at most $2\lceil \log_2(k+1) \rceil - 1$ variables. Thus, the function \hat{F} can be also represented with at most $2\lceil \log_2(k+1) \rceil - 1$ variables. ■

From the experimental results in Section VIII, we have the following:

Conjecture 3: When k minterms are selected randomly, in most cases, a multiple-output functions defined for only the k minterms can be represented by at most $p = 2\lceil \log_2(k+1) \rceil - 1$ variables.

V. ALGORITHM TO MINIMIZE THE NUMBER OF VARIABLES

In this part, we consider an algorithm to represent an incompletely specified index generation function $f : D \rightarrow \{0, 1, \dots, k\}$, where $D \subset B^n$ by using the least number of variables. To show the idea of the method, we use the following:

Example 7: Let us minimize the number of variables to represent the index generation function shown in Table II.

- 1) Let the four vectors be $\vec{a}_1 = (1, 0, 0, 1)$, $\vec{a}_2 = (1, 1, 1, 1)$, $\vec{a}_3 = (0, 1, 0, 1)$, and $\vec{a}_4 = (1, 1, 0, 0)$.
- 2) To distinguish \vec{a}_1 and \vec{a}_2 , either x_2 or x_3 is necessary. This derives the condition $x_2 \vee x_3$. In the same way, to distinguish \vec{a}_1 and \vec{a}_3 , we need $x_1 \vee x_2$; to distinguish \vec{a}_1 and \vec{a}_4 , we need $x_2 \vee x_4$; to distinguish \vec{a}_2 and \vec{a}_3 , we need $x_1 \vee x_3$; to distinguish \vec{a}_2 and \vec{a}_4 , we need $x_3 \vee x_4$; and to distinguish \vec{a}_3 and \vec{a}_4 , we need $x_1 \vee x_4$.
- 3) To distinguish all the vectors, all the conditions must hold at the same time. Thus, we have $R = (x_2 \vee x_3)(x_1 \vee x_2)(x_2 \vee x_4)(x_1 \vee x_3)(x_3 \vee x_4)(x_1 \vee x_4)$
- 4) By the distributive law, and the absorption law, we have $R = x_1 x_2 x_4 \vee x_1 x_2 x_3 \vee x_2 x_3 x_4 \vee x_1 x_3 x_4$.
- 5) Since every product has three literals, each corresponds to a minimum solution. Thus, f can be represented by three variables. (End of Example)

In principle, the above method produces the minimum number of variables to represent an incompletely specified index generation function. However, the straightforward application

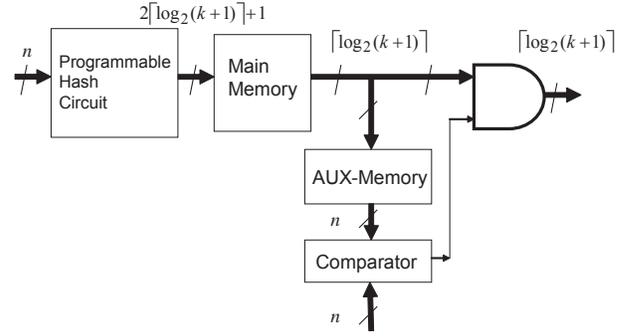


Fig. 4. A circuit for index generation function.

is quite inefficient. Also, we have an efficient minimization algorithm for SOPs, but do not have one for product-of-sums expressions. Thus, instead of obtaining R directly, first we obtain \bar{R} , the complement of R , and perform simplification, and then convert \bar{R} into the SOP for R as follows:

Algorithm 1: (Algebraic Method)

- 1) Let A be the set of vectors \vec{a}_i , such that $f(\vec{a}_i) = i$, where $i = 1, 2, \dots, k$
- 2) For each pair of vectors $\vec{a}_i = (a_1, a_2, \dots, a_n) \in A$ and $\vec{b}_j = (b_1, b_2, \dots, b_n) \in A$, associate a product defined by $s(i, j) = \bigwedge_{r=1}^n y_r$, where $y_r = 1$ if $a_r = b_r$ and $y_r = \bar{x}_r$ if $a_r \neq b_r$, where $r = 1, 2, \dots, n$. Note that there are $k(k-1)/2$ pairs.
- 3) Define a covering function $\bar{R} = \bigvee_{i < j} s(i, j)$.
- 4) Represent \bar{R} by the a minimum SOP.
- 5) Represent R , the complement of \bar{R} by a minimum SOP.
- 6) The product with the fewest literals corresponds to the minimum solution.

In Algorithm 1, Steps 4, 5, and 6 perform a minimum covering. In our implementation, we solve it by a sparse matrix data structure instead of the algebraic method. For each product s_j in Step 2, assign column j , and for each variable x_i assign a row i . The entry (i, j) in the covering table has 1 iff the product s_j has the literal x_i . Thus, the original covering table has n rows and $\frac{k(k-1)}{2}$ columns. Thus, the reduction of the number of columns is vitally important. Since the covering function \bar{R} is a negative unate, we generate only products that are not absorbed by other products. The resulting covering table is smaller than ones for minimization of SOPs [3].

VI. A CIRCUIT FOR INDEX GENERATION FUNCTIONS

Here, we consider methods to implement index generation functions. A software-based method is too slow for internet applications. A straightforward method is to use a programmable logic array (PLA) or content addressable memory (CAM). Unfortunately, PLAs and CAMs dissipate much more power and require more area than RAMs [11]. So, we consider a memory-based design.

Fig. 4 shows a circuit to implement an index generation function. The **programmable hash circuit** has n inputs and at most $2\lceil \log_2(k+1) \rceil - 1$ outputs. It is used to rearrange the *care*

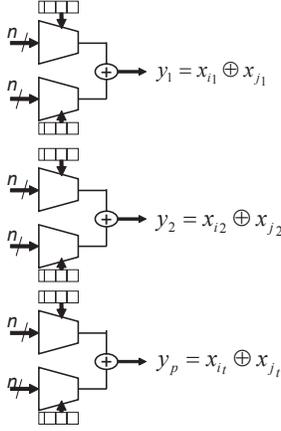


Fig. 5. Double-input hash circuit.

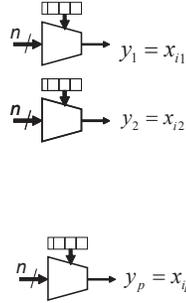


Fig. 6. Single-input hash circuit.

elements. We consider two types of programmable hash circuits. The first type is the **double-input hash circuit** shown in Fig. 5. It performs a linear transformation $y_i = x_i \oplus x_j$ or $y_i = x_i$, where $i \neq j$. It uses a pair of multiplexers for each variable y_i . The upper multiplexers have the inputs x_1, x_2, \dots, x_n . The lower multiplexers have the inputs x_1, x_2, \dots, x_n , except for x_i . For the i -th input, the constant input 0 is connected instead of x_i . By setting $y_i = x_i \oplus 0$, we can implement $y_i = x_i$. The second type of a programmable hash circuit is the **single-input hash circuit** shown in Fig. 6. It consists of only p multiplexers, and selects $p = 2^{\lceil \log_2(k+1) \rceil} - 2$ variables from n input variables. The **main memory** has at most $2^{\lceil \log_2(k+1) \rceil} - 1$ inputs and $\lceil \log_2(k+1) \rceil$ outputs. The main memory produces correct outputs only for registered vectors. However, it may produce incorrect outputs for non-registered vectors, because the number of input variables is reduced by using *don't care* conditions. In an index generation function, if the input vector is non-registered, then it should produce 0 outputs. To check whether the main memory produces the correct output or not, we use the **AUX memory**. The AUX memory has $\lceil \log_2(k+1) \rceil$ inputs and n outputs: It stores the registered vectors for each index. The **comparator** checks if the inputs are the same as the registered vector or not. If they are the same, the main memory produces a correct output. Otherwise, the main memory produces a wrong output, and the input vector is non-registered. Thus, the **output AND gates** produce 0 outputs, showing that the input vector is non-registered. Note that the main memory produces the correct outputs only for the registered vectors. In this way, we can implement an incompletely specified index generation function instead of a completely specified one² in the main memory. Let $m = \lceil \log_2(k+1) \rceil$. Then, the number of bits for the main memory is $m2^{2^m-1} \approx \frac{1}{2}m(k+1)^2$. The number of bits for the AUX memory is $n2^m \approx n(k+1)$. In many cases, $\frac{1}{2}km \gg n$, thus, the size of the AUX memory is much smaller than that of the main memory.

²The output AND, the AUX memory and the comparator are used to establish *observability don't cares* for the main memory.

VII. REDUCTION OF THE NUMBER OF VARIABLES BY A LINEAR TRANSFORMATION

As shown in Conjecture 2, most incompletely specified index generation functions with weight k can be represented by at most $p = 2^{\lceil \log_2(k+1) \rceil} - 1$ variables. However, there exist functions that require more variables. Example 4 shows such a function. In such a case, we can often reduce the number of variables by a linear transformation of the input variables.

Example 8: Consider the incompletely specified index generation function $f(x_1, x_2, x_3, x_4)$, where $f(0, 0, 0, 0) = 1$, $f(1, 0, 0, 0) = 2$, $f(0, 1, 0, 0) = 3$, $f(0, 0, 1, 0) = 4$, and $f(0, 0, 0, 1) = 5$. Note that all the variables are essential in f . Now, replace the variables x_1, x_2, x_3 , and x_4 with $y_1 = x_1 \oplus x_4$, $y_2 = x_2 \oplus x_4$, $y_3 = x_3$, and $y_4 = x_4$, respectively. Then, f can be represented as the index generation function $g(y_1, y_2, y_3, y_4)$ whose registered vectors are $(0, 0, 0, 0)$, $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, and $(1, 1, 0, 1)$. Note that g can be represented by using only y_1, y_2 , and y_3 , since they can uniquely specify 5 different patterns. The programmable hash circuit in Fig. 6 performs this linear transformation. (End of Example)

We have developed a heuristic algorithm to find a linear transformation that reduces the number of variables, when the double-input hash circuit is used. To find a linear transformation that reduces the maximum number of variables, we use the following:

Theorem 4: Let $f(x_1, x_2, \dots, x_n)$ be an incompletely specified index generation function with weight k . Let $Y_1 = (y_1, y_2, \dots, y_p)$, where $y_i = x_i \oplus x_j$ and $j \in \{p+1, p+2, \dots, n\}$, and $X_2 = (x_{p+1}, x_{p+2}, \dots, x_n)$. Consider the transformed function $g(Y_1, X_2) = f(X_1, X_2)$. Then, f can be represented by using only Y_1 , if and only if the column multiplicity of the decomposition table (Y_1, X_2) is $k+1$.

VIII. EXPERIMENTAL RESULTS

A. Random Single-Output Functions

For different n , we randomly generated 1000 functions, where k combinations are mapped to 0, k combinations are mapped to 1, and the other $2^n - 2k$ combinations are mapped to *don't cares*. We minimized the number of variables by an exact optimization algorithm, which is similar to Algorithm 1 shown in Section V. Table III shows the average numbers of variables to represent the single-output functions, where the set of variables are selected by the optimization algorithm. For example, 16-variables functions where 15 minterms are mapped to zeros, 15 minterms are mapped to ones, and the other minterms are mapped to *don't cares*, require, on the average, only 5.157 variables to represent the functions. Table III shows that the necessary number of variables to represent the functions mainly depends on k . The last column of the table shows the number of variables to represent incompletely specified functions by Conjecture 1. For example, when $k = 15$, to represent a uniformly distributed function, Conjecture 1 shows that 6 variables are sufficient. On the

TABLE III

AVERAGE NUMBERS OF VARIABLES TO REPRESENT SINGLE-OUTPUT LOGIC FUNCTIONS WITH k 1'S AND k 0'S.

k	$n = 16$	$n = 20$	$n = 24$	$2\lceil\log_2(k+1)\rceil - 2$
15	5.157	4.981	4.940	6
31	7.126	6.980	6.003	8
63	9.179	8.972	8.861	10
127	11.362	10.971	10.776	12
255	13.754	12.990	12.725	14
511	15.739	15.098	14.805	16
1023	16.000	17.508	16.918	18
2047	16.000	19.705	18.996	20
4095	16.000	20.000	21.394	22
8191	16.000	20.000	23.630	24

TABLE IV

AVERAGE NUMBER OF VARIABLES TO REPRESENT INCOMPLETELY SPECIFIED INDEX GENERATION FUNCTION.

k	$n = 16$	$n = 20$	$n = 24$	$2\lceil\log_2(k+1)\rceil - 1$ Conjecture 2
15	4.980	4.947	4.878	7
31	6.447	6.115	6.003	9
63	8.257	8.007	8.000	11
127	10.304	10.000	9.963	13
255	12.589	11.996	11.896	15
511	14.890	14.019	13.787	17
1023	15.991	16.293	15.874	19
2047	16.000	18.758	17.965	21
4095	16.000	19.992	20.093	23

other hand, experimental results show that only 4, 5, or 6 variables are necessary to represent the functions. We note that the variance is very small.

B. Random Index Generation Functions

We generated uniformly distributed index generation functions. Table IV shows the average numbers of variables to represent n -variables index generation functions with k registered vectors. For the other $2^n - k$ combinations, the outputs are set to *don't cares*. The values are the average of 1000 randomly generated functions. Table IV shows that the necessary number of variables to represent the functions strongly depends on k .

The last column of Table IV shows the number of variables to represent incompletely specified index generation functions with weight k given by Conjecture 2. For example, when $k = 31$, to represent a uniformly distributed function, Conjecture 2 shows that 9 variables are sufficient. On the other hand, experimental results show that only 6 or 7 variables are necessary to represent the functions. Again, the variance is very small.

C. IP Address Table

To verify the effectiveness of the method in the practical applications, we used distinct IP addresses of computers that accessed our web site over a period of a month. We considered four lists of different sizes: List 1, List 2, List 3, and List 4. Table V shows the results. The first row shows the number of registered vectors: k . The second row shows the number of inputs: n . The third row shows the number of outputs: $m =$

TABLE V

REALIZATION OF IP ADDRESS TABLES.

	List 1	List 2	List 3	List 4
# of vectors: k	1670	3288	4591	7903
# of inputs: n	32	32	32	32
# of outputs: m	11	12	13	13
$2\lceil\log_2(k+1)\rceil - 1$	21	23	25	25
# of variables using Single-input hash: n_s	18	20	21	23
# of variables using Double-input hash: n_d	17	19	20	21
Single-memory realization ($\times 10^{10}$ bits)	4.72	5.15	5.58	5.58
Realization using Single- input hash ($\times 10^6$ bits)	2.95	12.7	27.5	109.3
Realization using Double- input hash ($\times 10^6$ bits)	1.51	6.3	13.9	27.5
CPU Time (sec)	14.3	35.2	7.9	52.3

$\lceil\log_2(k+1)\rceil$. The fourth row shows the number of variables sufficient to represent the functions given by Conjecture 2, i.e., $2\lceil\log_2(k+1)\rceil - 1$. The fifth row shows the number of variables to represent the function, where the number of variables is minimized by Algorithm 1. In this case, selected input variables are connected to the main memory through the single-input hash circuit shown Fig. 6. The sixth row shows the number of variables to represent the function, where a linear transformation is used to reduce the number of variables. In this case, a double-input hash circuit shown in Fig. 5 is used. The seventh row shows the number of bits to represent the function by a single memory: $m2^n$. The eighth row shows the total number of bits to represent the function by using the single-input hash circuit shown in Fig. 6: $m2^{n_s} + n2^m$, where the first term denotes the size of the main memory, while the second term denotes the size of the AUX memory. The ninth row shows the total number of bits to represent the function by using the double-input hash circuit shown in Fig. 5: $m2^{n_d} + n2^m$. And, the last row shows the cpu time for Algorithm 1. We used a PC with an Intel Core 2 Duo Processor E6600, 2.4 GHz on Windows XP. As shown in Table V, the total amount of memory can be drastically reduced.

D. Benchmark Multiple-Output Functions

We reduced the number of variables for selected PLA benchmark functions [13]. Table VI shows the numbers of variables to represent benchmark functions (*bc0*, *chkn*, *in2*, *in7*, *intb*, and *vg2*) for different values of *care* minterms k . In the table, n denote the number of original input variables, m denotes the number of outputs, and q denotes the number of products in the PLA. Out of 2^n combinations, we randomly selected k different combinations as *care* minterms, and set other $2^n - k$ minterms to *don't cares*. Then, minimized the number of variables. From the table, we observe that the number of variables strongly depends on k , but virtually independent of n , m , q , and function name. Again, for these benchmark functions, the upper bounds on the number of products given by the Conjecture 3 are valid.

TABLE VI
NUMBER OF VARIABLES TO REPRESENT INCOMPLETELY SPECIFIED
MULTIPLE-OUTPUT PLA BENCHMARK FUNCTIONS.

k	$bc0$	$chkn$	$in2$	$in7$	$intb$	$vg2$	$2\lceil\log_2(k+1)\rceil - 1$
	$n=26$ $m=11$ $q=179$	$n=29$ $m=7$ $q=142$	$n=19$ $m=7$ $q=135$	$n=27$ $m=10$ $q=55$	$n=15$ $m=7$ $q=631$	$n=25$ $m=8$ $q=110$	Conjecture 3
15	4	4	4	4	5	4	7
31	6	5	6	6	7	6	9
63	8	7	7	8	8	7	11
127	9	9	8	8	10	9	13
255	11	10	9	10	12	11	15
511	12	12	10	12	14	13	17
1023	14	14	13	13	15	14	19

IX. CONCLUSION AND COMMENTS

In this paper, we have derived the number of variables to represent incompletely specified functions. For random functions with a small number of *care* minterms, the necessary numbers of variables only depend on the number of the *care* minterms, and the variance is very small. This applies for both single and multiple output functions. An index generation function with k registered vectors can be represented by at most $2\lceil\log_2(k+1)\rceil - 1$ variables, in most cases.

We have presented a method to implement incompletely specified index generation functions, instead of a completely specified ones. Furthermore, we have shown a method to reduce the number of variables by a linear transformation of the input variables. With these methods, we can implement an index generation function by memories with fewer inputs. Various methods exist to implement index generation functions [8], [9], [10]. Although the presented method may take more time to reconfigure, the circuit is simpler than [9], [10].

ACKNOWLEDGMENTS

This work was supported in part by a Grant in Aid for Scientific Research of the JSPS, and Knowledge Cluster Project of MEXT. The author thanks Prof. Jon T. Butler for discussion and Mr. M. Matsuura for experiment.

REFERENCES

- [1] F. M. Brown, *Boolean Reasoning: The logic of Boolean Equations*, Kluwer Academic Publishers, Boston, 1990.
- [2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
- [3] O. Coudert and J. Madre, "New ideas for solving covering problems," *Proc. DAC*, pp. 641-646, June 1995.
- [4] M. Fujita and Y. Matsunaga, "Multi-level logic minimization based on minimal support and its application to the minimization of look-up table type FPGAs," *ICCAD-91*, pp. 560-563, 1991.
- [5] C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a Boolean functions," *IEEE Trans. on Computers*, Vol. C-27, No. 11, pp. 1064-1068, Nov. 1978.
- [6] Y. Kambayashi, "Logic design of programmable logic arrays," *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 609-617, Sept. 1979.
- [7] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," *30th International Symposium on Multiple-Valued Logic*, pp. 91-97, Portland, Oregon, U.S.A., May 23-25, 2000.
- [8] T. Sasao, "Design methods for multiple-valued input address generators," (invited paper) *International Symposium on Multiple-Valued Logic (ISMVL-2006)*, Singapore, May 2006.
- [9] T. Sasao, "A Design method of address generators using hash memories," *IWLS-2006*, pp. 102-109, Vail, Colorado, U.S.A., June 7-9, 2006.
- [10] T. Sasao and M. Matsuura, "An implementation of an address generator using hash memories," *10th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools (DSD-2007)*, pp. 69-76, Lubeck, Germany, Aug. 27-31, 2007.
- [11] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, Vol. 37, Issue 3, pp. 238-275, Sept., 2005.
- [12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *ACM SIGCOMM Computer Communication Review*, Vol. 27, No. 4, pp. 25-38, 1997.
- [13] S. Yang, "Logic synthesis and optimization benchmark user guide, Version 3.0," *MCNC*, Jan. 1991.