Representation of Incompletely Specified Index Generation Functions Using Minimal Number of Compound Variables

Tsutomu Sasao, Takaaki Nakamura , Munehiro Matsuura
Kyushu Institute of Technology, Iizuka 820-8502, Japan

Abstract

This paper shows a method to reduce the number of input variables to represent incompletely specified index generation functions. A compound variable is generated by EXORing the original input variables. By using both original and compound variables, incompletely specified index generation functions can be represented by fewer variables. As a means to select variables, a heuristic method using information gains is presented. We compare representing random functions using 1. only original variables, and 2. both original and compound variables. Experimental results show that the use of compound variables effectively reduces the number of input variables.

1 Introduction

Consider an index generator that stores k different n-bit vectors, and produces an output of 0 if an n-bit input vector does not match the contents of any of the k vectors. If a match occurs, the output is the index of the matched vector.

Index generators are used in IP address lookup [12] and terminal access controllers [10]. To implement index generators, CAM (Content Addressable Memory) or PLA (Programmable Logic Array) can be used. However, power dissipation for these devices is relatively high [11]. Thus, implementations using ordinary memories are desirable. However, the single-memory realization of index generators is impractical, since n, the number of input variables, is often larger than 32. To reduce the memory size, a method using a main memory and an auxiliary memory has been developed [10]. In an index generator, the number of registered vectors k, is much smaller than the number of the possible input combinations, 2^n. In such a case, the size of the main memory is drastically reduced by reducing the number of input variables.

In this paper, we consider a reduction method of the input variables for incompletely specified index generation functions. In this case, the given function is realized by reduced-input memories and some extra circuits consisting of EXOR gates, registers and multiplexers. The rest of the paper is organized as follows: Section 2 defines incompletely specified index generation functions, and shows their properties. Section 3 shows a method to represent an incompletely specified index generation function using a minimal number of compound variables. Section 4 shows an index generator using memories. Section 5 shows a heuristic method to reduce the number of variables to represent incompletely specified index generation functions. Section 6 shows experimental results for the exact and heuristic methods. Finally, Section 7 summarizes the work. [1]

2 Incompletely Specified Index Generation Functions

Definition 2.1 *Let $D = \{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k\}$ be a set of k different vectors in B^n, where $B = \{0, 1\}$. Each vector in D is called a* **registered vector**. *The* **index table** *contains the corresponding index i for each vector $\vec{v}_i$, where $i \in \{1, 2, \ldots, k\}$.*
$f : B^n \rightarrow \{1, 2, \ldots, k\}$ is an **index generation function with weight** *k if*

$$\begin{aligned} f(\vec{a}_i) &= i, (when\ \vec{a}_i \in D), and \\ f(\vec{b}) &= 0, (otherwise), \end{aligned}$$

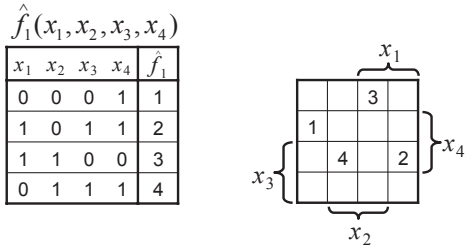
where $i \in \{1, 2, \ldots, k\}$.
$\hat{f} : B^n \rightarrow \{1, 2, \ldots, k, d\}$ is an **incompletely specified index generation function with weight** *k if*

$$\begin{aligned} f(\vec{a}_i) &= i, (when\ \vec{a}_i \in D), and \\ f(\vec{b}) &= d, (otherwise), \end{aligned}$$

where d denotes the don't care value.

The number of variables to represent incompletely specified logic functions can often be reduced [1, 2, 6, 10].

[1] The affiliation of the second author is currently, Elpida Memory Inc., Tokyo 104-0028, Japan.



(a) Index Table (b) Decomposition Table

Figure 2.1. Reduction of variables to represent an incompletely specified index generation function

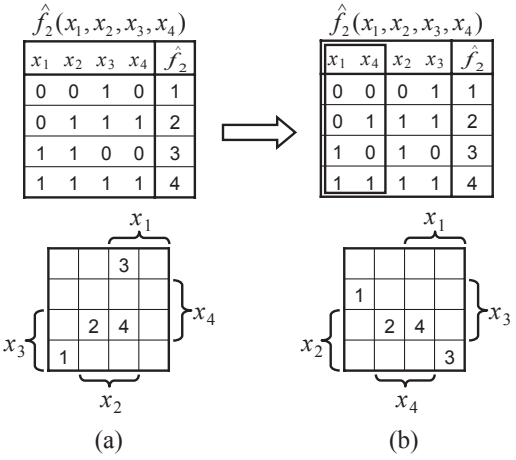


Figure 2.2. Reduction of variables to represent an input incompletely specified index generation function

Example 2.1 *Consider the index table shown in Fig. 2.1(a). The corresponding decomposition chart for the incompletely specified index generation function is shown in Fig. 2.1(b), where blank cells denote don't cares. In this function, for the vectors* $\vec{a}_1 = (0,0,0,1)$, $\vec{a}_2 = (1,0,1,1)$, $\vec{a}_3 = (1,1,0,0)$, *and* $\vec{a}_4 = (0,1,1,1)$, *the values of functions are* $\hat{f}_1(\vec{a}_1) = 1$, $\hat{f}_1(\vec{a}_2) = 2$, $\hat{f}_1(\vec{a}_3) = 3$, *and* $\hat{f}_1(\vec{a}_4) = 4$, *respectively. For other inputs, the values of* $\hat{f}_1$ *are* d *(don't care or undefined).*

In the decomposition chart, when each column has at most one specified element, then the function can be represented by column variables only, since for each column, all don't cares values can be set to the specified value in that column. In Fig. 2.1(a), values for (x_1, x_2) *are distinct, and the index can be specified by using only these two variables.* *(End of Example)*

As shown in the above example, in the decomposition chart, when each column has at most one specified element, the function can be represented by using only column variables.

Example 2.2 *Consider the index table in Fig. 2.2, and the decomposition chart for an incompletely specified index generation function* $\hat{f}_2$. *Consider the number of variables to represent the function. In the decomposition chart in Fig. 2.2(a), two non-zero elements exist in the column* $(x_1, x_2) = (1,1)$. *Thus, the function* $\hat{f}_2$ *cannot be represented by* $\{x_1, x_2\}$. *Similarly, in the row* $(x_3, x_4) = (1,1)$, *two non-zero elements exist, and the function* $\hat{f}_2$ *cannot be represented by* $\{x_3, x_4\}$, *either.*

Next, let us change the partition of the input variables into (x_1, x_4) *and* (x_2, x_3) *as shown in Fig. 2.2(b). In this case, each column has at most one specified element. Note that in the index table in Fig. 2.2(b), values of the vectors* (x_1, x_4) *are all different. Thus, the function* $\hat{f}_2$ *can be represented by using only* $\{x_1, x_4\}$. *(End of Example)*

As shown in the above examples, to represent incompletely specified index generation functions, input variables can be often reduced. Minimization methods of input variables for single-output incompletely specified functions are considered in [1, 2].

Lemma 2.1 *Let* p *be the number of variables to represent an incompletely specified index generation function with weight* k. *Then, we have the following relation:*

$$p \geq \lceil \log_2(k+1) \rceil.$$

For example, let $k+1 = 2^p$. Consider the complete binary tree of height p. The k registered vectors can be distinguished by using p variables.

The minimization of the variables for an incompletely specified index generation function can be done by reducing the height of the binary tree representing the registered vectors. To reduce the height of the tree, the variables should be selected so as to make the height of the subtree as equal as possible. That is, the variables should be selected so as to make a balanced tree. This idea will be used in a heuristic algorithm to reduce the number of input variables as explained later. Experimental results show that, most incompletely specified index generation functions with weight k, can be represented by $2\lceil \log_2(k+1) \rceil - 1$ or fewer variables [10].

Conjecture 2.1 *Let* p *be the number of variables to represent an incompletely specified index generation function with weight* k. *Then, for most index generation functions, we have the following relation:*

$$p \leq 2\lceil \log_2(k+1) \rceil - 1.$$

3 Representation of Index Generation Functions Using Compound Variables

In this part, we show a method to reduce the number of variables to represent an incompletely specified function by using compound variables.

Definition 3.1 *For n input variables $\{x_1, x_2, \ldots, x_n\}$, a* **compound variable** *y has a form*

$$y = c_1x_1 \oplus c_2x_2 \oplus \cdots \oplus c_nx_n,$$

where $c_i \in \{0, 1\}$. The **compound degree** *of y is $\delta = \sum_{i=1}^{n} c_i$. A variable with compound degree 1 is a* **primitive variable**. *A variable with compound degree 2 is a* **bi-compound variable**, *and a variable with compound degree 3 is a* **tri-compound variable**.

Example 3.1 *Consider the incompletely specified index generation function $\hat{f}_3$ shown in Fig. 3.1. Consider the number of variables to represent this function. In Fig. 3.1(a), the column $(x_1, x_2) = (1, 1)$ has two non-zero elements. So, the function cannot be represented by $\{x_1, x_2\}$. In a similar way, the row $(x_3, x_4) = (1, 1)$ has two non-zero elements. So, the function cannot be represented by $\{x_3, x_4\}$. Note that the decomposition chart with other partitions produce the same results. Thus, to represent the function $\hat{f}_3$, at least three variables are necessary. Next, consider the bi-compound variables $y_1 = x_1 \oplus x_2$ and $y_2 = x_2 \oplus x_3$. In this case, we have the function $\hat{g}_3(y_1, y_2, x_3, x_4)$ shown in Fig. 3.1(b). Note that, in the decomposition chart shown in Fig. 3.1(b), each column has at most one specified element. Thus, the function $\hat{g}_3$ can be represented by using only two variables $\{y_1, y_2\}$.* *(End of Example)*

As shown in the above example, by using compound variables, the number of input variables for incompletely specified index generation functions can be further reduced. In the rest of the paper, both a primitive variable x_i and a compound variables y_j are treated as input variables.

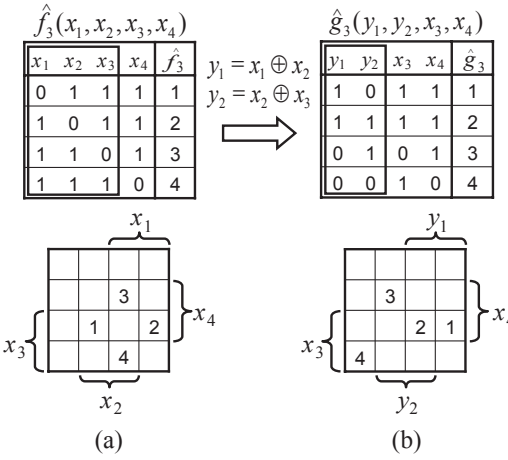


Figure 3.1. Incompletely specified index generation function represented by compound variables.

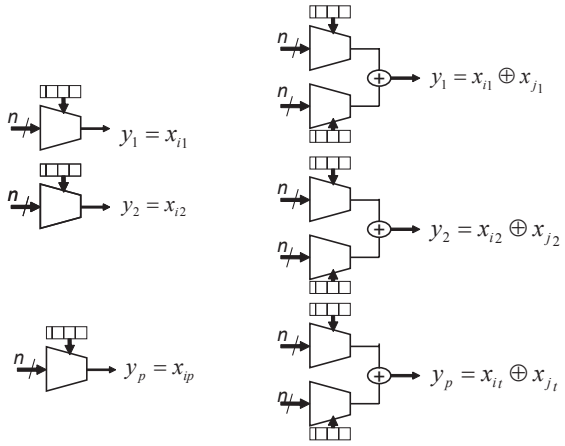


Figure 3.2. Primitive variable generator.

Figure 3.3. Bi-compound variable generator.

4 Index Generator

Fig. 4.1 is an index generator using two memories[10]. 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* elements. This corresponds to compound variable generators. We consider three types of programmable hash circuits.

The first type generates **primitive variables** as shown in Fig. 3.2. It consists of p multiplexers, and selects variables from n input variables. When only primitive variables are used, the circuit in Fig. 3.2 can be used.

The second type generates **bi-compound variables** as shown in Fig. 3.3. 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 $x_1, x_2, \ldots, x_n$. The lower multiplexers have the inputs $x_1, x_2, \ldots, x_n$, 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 values of control variables for the multiplexers are stored in the registers. Thus, an arbitrary variable can be selected from n input variables. Fig. 3.3 generates bi-compound variables.

The third type generates tri-compound variables. For each output, it requires three multiplexers and a three-input

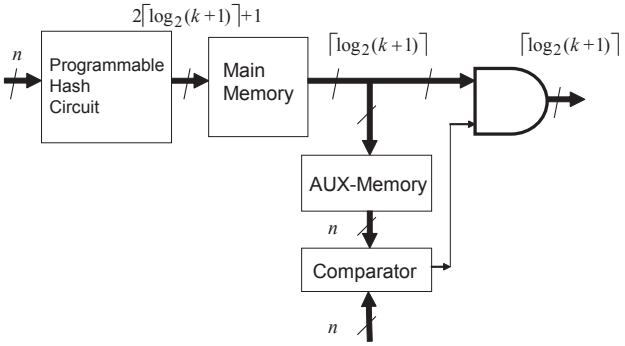


Figure 4.1. Index generator using two memories.

EXOR gate (the figure is omitted).

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. In this case, 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 one [2] in the main memory. Let $p = \lceil\log_2(k+1)\rceil$. Then, the number of bits in the main memory is at most $p2^{2p-1} \approx \frac{1}{2}p(k+1)^2$. The number of bits in the AUX memory is $n2^p \approx n(k+1)$. In many cases, $\frac{1}{2}kp >> n$, thus, the size of the AUX memory is much smaller than that of the main memory.

5 Methods to Select Compound Variable

When only primitive variables are used, the number of variables for an incompletely specified index generation function can be minimized by solving a kind of a minimum covering problem [6, 10].

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

Algorithm 5.1 *(Exact minimization of the number of variables to represent an incompletely specified index generation function)*

1. *Represent the conditions to distinguish the pairs of registered vectors by logical expressions.*
2. *Obtain the minimum weight assignments to the values that cause the values of logical expressions to be 1.*

Example 5.1 *Let us minimize the number of variables to represent the incompletely specified index generation function $\hat{f}_2$ shown in Fig. 2.2(a).*

1. *Let the four registered vectors be $\vec{a}_1 = (0,0,1,0)$, $\vec{a}_2 = (0,1,1,1)$, $\vec{a}_3 = (1,1,0,0)$, and $\vec{a}_4 = (1,1,1,1)$.*
2. *To distinguish $\vec{a}_1$ and $\vec{a}_2$, either x_2 or x_4 is necessary. Thus, we have the condition $x_2 \vee x_4 = 1$. Similarly, to distinguish $\vec{a}_1$ and $\vec{a}_3$, we have the condition $x_1 \vee x_2 \vee x_3 = 1$; to distinguish $\vec{a}_1$ and $\vec{a}_4$, we have the condition $x_1 \vee x_2 \vee x_4 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_3$, we have the condition $x_1 \vee x_3 \vee x_4 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_4$, we have the condition x_1=1; and to distinguish $\vec{a}_3$ and $\vec{a}_4$, we have the condition $x_3 \vee x_4 = 1$.*
3. *To distinguish all the registered vectors, all the conditions must be true at the same time. Thus, we have the condition $R = 1$, where $R = x_1(x_2 \vee x_4)(x_3 \vee x_4)(x_1 \vee x_2 \vee x_3)(x_1 \vee x_2 \vee x_4)(x_1 \vee x_3 \vee x_4)$.*
4. *By the distributive law and the absorption law, we have the relation $R = x_1x_2x_3 \vee x_1x_4$. The degree of the product term with the minimum number of literals is two. The product term x_1x_4 shows the minimum solution. Thus, the function $\hat{f}_2$ can be represented by two variables x_1 and x_4.*

(End of Example)

In principle, the minimization of variables consisting of both primitive and compound variables can be done in the same way as Algorithm 5.1. That is, we can perform the minimization of the variables, where not only the primitive variables $x_1, x_2, \ldots, x_n$, but also the compound variables $y_1, y_2, \ldots, y_m$ can be considered as the input variables. When both the primitive and the bi-compound variables are used, the number of the input variables to consider is

$$n + \binom{n}{2} = \frac{n(n+1)}{2}.$$

When tri-compound variables in addition to the bi-compound and the primitive variables are used, the number of the variables to consider is

$$n + \binom{n}{2} + \binom{n}{3} = \frac{n(n^2+5)}{6}.$$

This problem can be solved by first representing the logical expressions by a BDD (Binary Decision Diagram), and then finding the path with the minimum weight. When we use a BDD to minimize the number of variables, the size of the BDD increases exponentially with the number of the input variables. Thus, only a limited number of input variables can be solved.

Definition 5.1 *In an incompletely specified index generation function with weight k, the* **balance factor** *with respect to x_i is $k - |h_{i0} - h_{i1}|$, where h_{i0} is the number of registered vectors such that $x_i = 0$, and h_{i1} is the number of registered vectors such that $x_i = 1$.*

From here, we will consider a heuristic method to represent incompletely specified functions. Variables with as large balance factor tend to partition the set of vectors into more balanced sets. Let k be the number of registered vectors. When the given set of variables partitions the set of vectors into balanced sets, the number of variables to represent the function is reduced to $\lceil \log_2(k+1) \rceil$. From this, we have the following:

Algorithm 5.2 *(Heuristic reduction of the number of variables)*

1. *Let the input variables be $x_1, x_2, \ldots, x_n$.*
2. *Select m compound variables y_i. In this case, select variables with as large a balance factor as possible.*
3. *By using Algorithm 5.1, reduce the number of the input variables for an incompletely specified index generation function, where the number of original inputs variables is $n + m$, and the number of vectors is k.*

The larger the value of m, the better solutions we can get. However, computation time increases with an increase of m.

Next, we present the **information gain method**, a heuristic method to select compound variables. The selection of the compound variables can be considered as the optimization of the binary decision tree [3].

When selecting compound variables, the larger the balance factor of a variable, the larger the information gain we can achieve. Thus, a variable with a large information gain (i.e., a variable with a large balance factor) tends to reduce the number of variables in the decision tree.

Algorithm 5.3 *(The information gain method)*

1. *Among the compound variables and primitive variables, select a variable y_1 with the largest balance factor. Then, partition the registered vectors into two sets with $y_1 = 0$ and $y_1 = 1$. Let $i \leftarrow 1$.*
2. *Among the sets with more than one element, select a variable y_{i+1} whose minimal balance factor is the maximum, and partition the sets with y_{i+1}. Let $i \leftarrow i + 1$.*
3. *Iterate the above step until all the sets have one element.*
4. *The function can be represented by $(y_1, y_2, \ldots, y_p)$.*

Example 5.2 *Consider the incompletely specified index generation function $\hat{f}_4$ shown in Fig. 5.1. Select the compound variables by using the information gain method. Instead of the set of vectors, the algorithm is illustrated by the set of indices of the vectors.*

First, to partition the initial set $S_{11} = \{1, 2, 3, 4, 5, 6, 7\}$, the variable y_1 is selected. In this case, when $y_1 = x_1$, we have $h_0 = 4$, $h_1 = 3$ in the set S_{11}. Thus, the variable y_1 partitions the set S_{11} into $S_{21} = \{3, 4, 5, 6\}$ and $S_{22} = \{1, 2, 7\}$.

Second, two sets S_{21} and S_{22} are partitioned by the variable y_2. In this case, when $y_2 = x_2 \oplus x_3$, the set S_{21} is partitioned into $S_{31} = \{4, 5\}$ and $S_{32} = \{6, 3\}$. Also, the set S_{22} is partitioned into $S_{33} = \{7\}$ and $S_{34} = \{1, 2\}$.

Third, by using the variable y_3, S_{31}, S_{32}, and S_{34}, the sets with more than two elements, are partitioned. In this case, by selecting $y_3 = x_4$, each set is partitioned into two. After this partition, each set has just one element, and we terminate the algorithm. Thus, the function can be represented by $(y_1, y_2, y_3) = (x_1, x_2 \oplus x_3, x_4)$.
(End of Example)

6 Experimental Results

6.1 Design of the Experiment

To show the effectiveness of the algorithms, we minimized the number of variables for randomly generated incompletely specified index generation functions, where

1. only primitive variables are used ($\delta = 1$),
2. both primitive and bi-compound variables are used ($\delta \leq 2$), and
3. tri-compound variables in addition to primitive and bi-compound variables are used ($\delta \leq 3$).

When only primitive variables are used, the exact minimum solutions were obtained by using Algorithm 5.1. When compound variables are used, Algorithm 5.3 (information gain method) was used to obtain near-optimal solutions. Furthermore, to examine the optimality of the heuristic solutions, for small-scale problems that use bi-compound

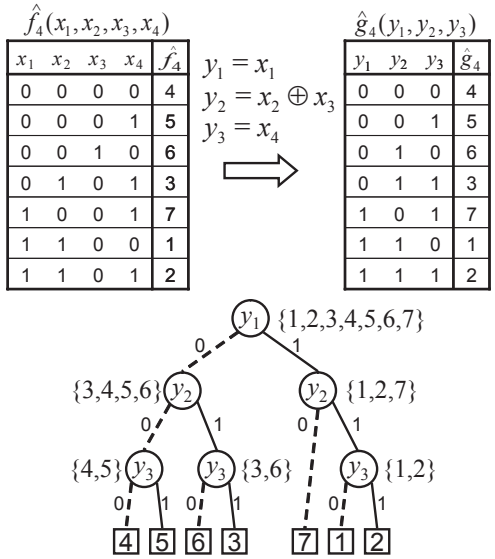


Figure 5.1. Selection of variables using information gain method

variables, optimal solutions were obtained by Algorithm 5.1.

For all combinations of (k, s), where k = 15, 63, 255, 1023 and $s = 0, 5, 10$, we randomly generated 1000 incompletely specified index generation functions of $n = 24$ variables, where s is the **skew factor** showing the distribution of 0's and 1's in the registered vectors. When $s = 0$, the 0's and 1's appear in the equal probability. A large s denotes a high probability of 0's appearing in the registered vectors. The registered vectors are generated as follows:

Algorithm 6.1 *(Generation of Registered Vectors)*

1. *Let the threshold be* $Th = (2^{30} - 1) + (2^{26} \times s)$.
2. *Obtain a random number* R. *Let* R_1 *be the integer represented by the least significant 31 bits of* R. *If* $R_1 \geq Th$, *generate 1. Otherwise, generate 0.*
3. *Perform Step 2* n *times to generate an* n*-bit vector.*
4. *Perform Step 3* k *times to generate* k *registered vectors. When the identical vectors appear, ignore it.*

To assess the quality of the solutions, we use the excessive variable ratio:

Definition 6.1 *Consider an* n*-variable incompletely specified index generation function with* k *registered vectors. Let* $\hat{n}$ *be the number of input variables after reduction. Then, the* **excessive variable ratio** *is*

$$r_e = \frac{\hat{n}}{\lceil \log_2(k+1) \rceil} - 1 \qquad (6.1)$$

$r_e = 0$ *shows that an optimal solution is achieved. A large* r_e *shows that the representation uses many extra variables.*

6.2 Reduction of Input Variables

Table 6.1 shows the average number of variables after reduction and the excessive variable ratio r_e, for different values of k, s, and δ, where $n = 24$.

6.2.1 Influence of Skew Factor

$s = 0$ denotes that the probabilities 0's and 1's appearing in the registered vectors are the same, while large s denotes that 0's appears more often than 1's. When k and δ are fixed, functions with large s require more variables.

6.2.2 Influence of Compound Degree

Variables with large δ require fewer variables to represent the function. Especially for functions with many registered vectors and a large skew factor s, variables with large δ are necessary to achieve small r_e. For example, when k=1023 and $s = 10$, representation of the functions using only primitive variables require 24 variables. This means that we cannot reduce the number of variables. We conjecture that representations with variables with $\delta \geq 4$ require fewer variables.

6.2.3 Computation Time

Table 6.2 compares the computation time of Algorithm 5.1 (exact) with that of Algorithm 5.3 (heuristic). Algorithm 5.1 minimizes the number of primitive variables, while Algorithm 5.3 minimizes the number of both primitive and compound variables. Note that Algorithm 5.3 spent much less time than Algorithm 5.1. This shows that the heuristic method is quite fast.

Algorithm 5.1 took up to 575.171 seconds to select primitive and bi-compound variables when $s = 10$. On the other hand, Algorithm 5.3 took at most 0.047 seconds. In the experiment, we used a computer with Core 2 Duo 2.66GHz, 4GByte of memory, running on Windows XP Professional Service Pack 3.

6.3 Comparison of Optimal Solutions with Heuristic Ones

To see the optimality of heuristic solutions obtained by Algorithm 5.3, we obtained the exact optimum solutions by Algorithm 5.1, and calculated the relative error. Table 6.3 compares optimal solutions (A) with heuristic solutions (B). The relative error is calculated as $\frac{B-A}{A}$. When both primitive and bi-compound variables are used, the relative error was at most 0.118.

Table 6.1. Numbers of variables and r_e

k	s	δ	Average # of variables	r_e
15	0	1	4.882	0.221
		≤ 2	4.209	0.052
		≤ 3	4.000	0.000
	5	1	4.997	0.249
		≤ 2	4.299	0.075
		≤ 3	4.000	0.000
	10	1	6.432	0.608
		≤ 2	4.905	0.226
		≤ 3	4.059	0.015
63	0	1	7.996	0.333
		≤ 2	7.968	0.328
		≤ 3	7.334	0.222
	5	1	8.936	0.489
		≤ 2	7.983	0.331
		≤ 3	7.381	0.230
	10	1	13.480	1.247
		≤ 2	9.448	0.575
		≤ 3	7.966	0.328
255	0	1	11.852	0.482
		≤ 2	11.819	0.477
		≤ 3	11.000	0.375
	5	1	13.212	0.652
		≤ 2	12.001	0.500
		≤ 3	11.003	0.375
	10	1	21.952	1.744
		≤ 2	15.543	0.943
		≤ 3	12.240	0.530
1023	0	1	15.889	0.589
		≤ 2	15.921	0.592
		≤ 3	15.016	0.502
	5	1	18.248	0.825
		≤ 2	16.351	0.635
		≤ 3	15.021	0.502
	10	1	24.000	1.400
		≤ 2	20.395	1.040
		≤ 3	16.027	0.603

k: number of registered vectors, s: skew factor, δ: compound degree; r_e: excessive variable ratio, $n = 24$, when $\delta = 1$ Algorithm 5.1 was used, when $\delta \leq 2$ or $\delta \leq 3$ Algorithm 5.3 was used.

Table 6.2. Average CPU time

k	δ	Algorithm	Average CPU time [sec]
63	1	5.1	1.454
	≤ 2	5.3	0.002
	≤ 3	5.3	0.014
255	1	5.1	7.647
	≤ 2	5.3.	0.010
	≤ 3	5.3	0.079
1023	1	5.1	1.011
	≤ 2	5.3	0.056
	≤ 3	5.3	0.450

δ: compound degree; $s = 0$, $n = 24$.

Table 6.3. Quality of heuristic solutions

s	Optimal A	Heuristic B	Relative error $(B - A)/A$
0	4.000	4.033	0.008
5	4.000	4.088	0.022
10	4.339	4.851	0.118

s: skew factor. $n = 24$, $k = 15$.
For optimal solutions, Algorithm 5.1 was used.
For heuristic solutions, Algorithm 5.3 was used.

7 Conclusion and Comments

In this paper, we presented methods to reduce the number of variables to represent incompletely specified index generation functions. Also, we demonstrated that the number of variables can be further reduced by using compound variables. Since the exact minimization of the variables is time-consuming, we developed a heuristic algorithm. Experimental results show the effectiveness of the method.

The circuit to realize tri-compound variables may be too expensive for some applications. For such applications, a method using both primitive and bi-compound variables seems to be promising.

In this paper, we used linear functions as compound variables, since they can be realized by simple circuits, and have been shown to be cost effective by experiments. Also, linear-transformations are well studied [5], and implementation is relatively easy. However, in principle, compound variables can be any functions. For example, instead of using the bi-compound variable generator shown in Fig. 3.3, 2-input LUTs can be used. This may further reduce the number of the input variables. Unfortunately, LUTs are more expensive than linear function generators, and no efficient algorithm to find the best compound variables is known.

8 Acknowledgments

This research is partly supported by the Japan Society for the Promotion of Science (JSPS) Grant in Aid for Scien-

tific Research, and the MEXT Knowledge Cluster Initiative (the second-stage). Discussion with Prof. Jon T. Butler was quite useful.

References

[1] 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.

[2] Y. Kambayashi, "Logic design of programmable logic arrays," *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 609-617, September, 1979.

[3] B. M. E. Moret, "Decision trees and diagrams," *ACM Computing Surveys*, Vol. 14, No. 4, pp. 594-623, December, 1982.

[4] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A Parallel sieve method for a virus scanning engine," *11th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece (DSD 2009).

[5] R. J. Lechner," Harmonic analysis of switching functions," in A. Mukhopadhyay (e.d.), *Recent Development in Switching Theory*, Academic Press, 1971.

[6] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," *International Symposium on Multiple-Valued Logic (ISMVL-2000)*, pp. 91-97, May, 2000.

[7] T. Sasao, "Design methods for multiple-valued input address generators," (invited paper) *International Symposium on Multiple-Valued Logic* (ISMVL-2006), Singapore, May 2006.

[8] 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.

[9] 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), Aug. 27 - 31, 2007, Lubeck, Germany, pp.69-76.

[10] T. Sasao, "On the numbers of variables to represent sparse logic functions," *International Conference on Computer Aided Design (ICCAD-2008)*, pp. 45-51, November, 2008.

[11] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, Vol. 37, Issue 3, pp. 238-275, September, 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.