An Algorithm to Find Optimum Support-Reducing Decompositions for Index Generation Functions

Tsutomu Sasao, Kyu Matsuura, and Yukihiro Iguchi
Dept. of Computer Science, Meiji University
Kawasaki 214-8571, Japan

***Abstract*—Index generation functions are useful for pattern matching. This paper presents an algorithm to find support-reducing decompositions for index generation functions. Let n be the number of the input variables, and let s be the number of bound variables. Then, the exhaustive search for finding an optimum support-reducing decomposition requires to check $\binom{n}{s}$ combinations. We found a special property of index generation functions that drastically reduces this search space. With this property, we developed a fast algorithm. For a given number of bound variables, it finds a decomposition with the fewest rails. Experimental results up to $n = 60$ and $s = 33$ are shown.**

I. Introduction

Index generation functions [10], [11] are useful for access control lists, routers, and virus scanning for the internet, etc.. Given a set of distinct k registered vectors of n bits, an index generation function produces an index of the corresponding vector. If the input vector is not registered, the function produces zero. The number of registered vectors is called a weight. Such function can be directly implemented by a content addressable memory (CAM). However, the cost and power dissipation of a CAM is much higher than a conventional memory [6].

An index generation function can be efficiently implemented by a look-up table (LUT) or an index generation unit (**IGU**), which are programmable [10]. An IGU consists of conventional memories and a small amount of logic. Thus, an IGU is much cheaper and dissipates lower power than a CAM.

Suppose that large-scale integrations (LSIs) for IGUs with n inputs and weight k are available. For a function with larger k, we can partition the set of vectors into several groups, and implement each group by an independent LSI. The outputs of the LSIs can be combined by an OR gate to produce the final output [12]. This is a **parallel decomposition**. For a function with larger n, we can partition the set of the input variables into two sets X_1 and X_2, and implement the function by a **serial decomposition** as shown in Fig. 1.1. In general, an optimum functional decomposition for an ordinary logic function is difficult to obtain. In this paper, for index generation functions, we show that an optimum decompositions can be obtained fairly efficiently.

The rest of the paper is organized as follows: Section II introduces decompositions. Section III introduces index generation functions. Section IV shows an efficient method to find an optimum support-reducing decomposition. Section V shows experimental results. Section VI concludes the paper.

II. Functional Decomposition

Functional decomposition [1], [3] is a technique to decompose a circuit into two parts with a lower cost than the original circuit. They are routinely used in logic synthesis [9].

Definition 2.1: [1] Let $f(X)$ be a function, and (X_1, X_2) be a partition of the input variables, where $X_1 = (x_1, x_2, \ldots, x_s)$ and $X_2 = (x_{s+1}, x_{s+2}, \ldots, x_n)$. The **decomposition chart** for f is a two-dimensional matrix with 2^s columns and 2^{n-s} rows, where each column and row is labeled by a unique binary assignment of values to the variables. The function f maps each assignment to $\{0, 1, \ldots, k\}$. The function represented by a column is a **column function** and is dependent on X_2. Variables in X_1 are **bound variables**, while variables in X_2 are **free variables**. In the decomposition chart, the **column multiplicity**, denoted by $\mu(f : X_1)$, is the number of different column functions. The set of bound variables is the **bound set**.

Example 2.1: Fig. 2.1 shows a decomposition chart of a 4-variable switching function. $X_1 = (x_1, x_2)$ denotes the bound variables, and $X_2 = (x_3, x_4)$ denotes the free variables. Since all the column patterns are different and there are four of them, the column multiplicity is $\mu(f : X_1) = 4$. ∎

Theorem 2.1: [3] For a given function f, let X_1 be the bound variables, let X_2 be the free variables, and let $\mu(f : X_1)$ be the column multiplicity of the decomposition chart. Then, the function f can be represented as $f(X_1, X_2) = g(h(X_1), X_2)$, and is realized with the network shown in Fig. 1.1. The number of signal lines connecting blocks H and G is $r = \lceil \log_2 \mu(f : X_1) \rceil$, where H and G realize h and g, respectively.

Definition 2.2: Consider the decomposition shown in Fig. 1.1. The signal lines connecting H and G are called **rails**. When the number of rails r is smaller than the number of the input variables in X_1, then function has a **support-reducing**

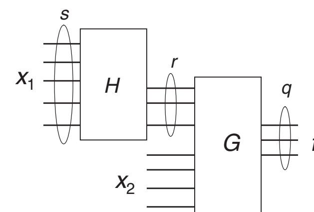

Fig. 1.1. Realization of a logic function by decomposition.

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

Fig. 2.1. Decomposition chart of an logic function.

		0	0	0	0	1	1	1	1	x_1
		0	0	1	1	0	0	1	1	x_2
		0	1	0	1	0	1	0	1	x_4
0	0	0	0	0	2	0	0	1	0	
0	1	0	0	0	0	0	5	0	0	
1	0	0	0	0	3	0	0	4	0	
1	1	0	0	0	0	0	6	7	0	
x_3	x_5									

Fig. 3.1. Decomposition chart for f.

TABLE 3.1
INDEX GENERATION FUNCTION

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

decomposition [4].

Logic functions for H and G can be realized by memories, and the complexities for G and H can be measured by the number of bits in the memories. Since an s-input r-output LUT stores $2^s r$ bits, we have the following:

Definition 2.3: Let s and r be the numbers of inputs and outputs of an LUT, respectively. Then, the size of the LUT is $r2^s$.

When the function has a support-reducing decomposition, the total amount of memory can often be reduced by realizing the circuit in Fig. 1.1 [10]. For the decomposition with $r = 1$, efficient algorithms using BDDs [2], [5], and SOP [7] are available. However, for $r \geq 2$, no efficient algorithm is available. In the later sections, we will show that for an index generation function, an optimum decomposition can be obtained efficiently for $r \geq 2$.

III. INDEX GENERATION FUNCTIONS

In this part, we define index generation functions and show their properties.

Definition 3.1: 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, for each registered vector, its **index**. An **index generation function** f produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. An index generation function represents a mapping: $f : B^n \rightarrow \{0, 1, 2, \ldots, k\}$, where $B = \{0, 1\}$. Typically, k is much smaller than 2^n, the total number of the input combinations.

Example 3.1: Consider the registered vector table shown in Table 3.1. It shows a 5-variable index generation function $f(X)$ with weight 7. Consider the decomposition chart for f shown in Fig. 3.1. $X_1 = (x_1, x_2, x_4)$ denotes the bound variables, and $X_2 = (x_3, x_5)$ denotes the free variables. Note that the column multiplicity of this decomposition chart is four. ■

Lemma 3.1: Let $\mu(f : X_1)$ be the column multiplicity of a decomposition chart of an index generation function $f(X_1, X_2)$. Let k be the weight of f, and s be the number of variables in X_1. Then,

$$\mu(f : X_1) \leq \min\{2^s, k + 1\}.$$

Lemma 3.2: Let f be an index generation function with weight k. Then, there exists a functional decomposition $f(X_1, X_2) = g(h(X_1), X_2)$, where g and h are index generation functions, the weight of g is k, and the weight of h is at most k.

(Proof) Consider a decomposition chart, in which X_1 denotes the bound variables, and X_2 denotes the free variables. Let $X_1 = (x_1, x_2, \ldots, x_s)$, where $s \geq \lceil \log_2(k + 1) \rceil$. Let h be a function where the variables are X_1, and the output values are defined as follows: Consider the decomposition chart, where assignments of values to X_1 label columns (i.e., bound variables). For the assignments to X_1 corresponding to columns with only zero elements, $h = 0$. For other inputs, the outputs are distinct integers from 1 to w_h, where w_h denotes the number of columns that have non-zero element(s). Since $w_h \leq k$, the weight of h is at most k, and the number of output values of h is at most $k + 1$. On the other hand, the function g is obtained from f by reducing some columns that have all zero output in the decomposition chart. Thus, the number of non-zero outputs in g is equal to the number of non-zero outputs in f. Thus, g is also an index generation function with weight k. □

Example 3.2: Consider the decomposition chart in Fig. 3.1. Let the function $f(X)$ be decomposed as $f(X_1, X_2) = g(h(X_1), X_2)$, where $X_1 = (x_1, x_2, x_4)$, while $X_2 = (x_3, x_5)$. Table 3.2 shows the function h. It can be considered as a 3-variable index generation function with weight 4. The decomposition chart for the function g is shown in Fig. 3.2. As shown in this example, the functions obtained by decomposing the index generation function f are also index generation functions, and the weights of f and g are both 7. ■

Theorem 3.1: Consider an n variable index generation function f with weight k. When $k \leq 2^{n-4} - 1$, f has a support-reducing decomposition, and its amount of memory is at most a half of the single-LUT realization.

TABLE 3.2
TRUTH TABLE FOR h.

x_1	x_2	x_4	y_1	y_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	0	0

		0	0	1	1	y_1
		0	1	0	1	y_2
0	0	0	1	2	0	
0	1	0	0	0	5	
1	0	0	4	3	0	
1	1	0	7	0	6	
x_3	x_5					

Fig. 3.2. Decomposition chart for g.

(Proof) Consider the decomposition $f(X_1, X_2) = g(h(X_1), X_2)$. By Lemma 3.1, the column multiplicity μ satisfies the relation $\mu \leq k+1$. Let r be the number of rails. From Theorem 2.1 and Lemma 3.1, we have the relation $r \leq q = \lceil \log_2(k+1) \rceil$. Then, the size of the LUT for H is at most $q \cdot 2^s$, where s denote the number of the bound variables, and $q = \lceil \log_2(k+1) \rceil \leq n-4$. The LUT for G requires $q \cdot 2^{q+(n-s)}$ bits. Thus, the total size is at most $q \cdot (2^s + 2^{q+n-s})$. This value takes its minimum when $s = q+n-s$. The minimum total size is $q \cdot 2^{s+1}$, and the original size is $q \cdot 2^n$. Thus, the ratio is $2^{n-s-1} \geq 2$. □

IV. OPTIMUM SUPPORT-REDUCING DECOMPOSITION

A. Exhaustive Method

In Definition 2.1, we specified that the first s variables $x_1, x_2, \ldots, x_s$ are in X_1, and the remaining $n-s$ variables are in X_2. However, in many cases, we can select any s variables for X_1. Unlike previous sections, X denotes un-ordered set, for simplicity.

Definition 4.1: Given an index generation function f, and given the number of bound variables s, the functional decomposition that minimizes the the number of rails $r < s$ is **an optimum support-reducing decomposition**.

Let n be the number of the input variables, and s be the number of bound variables. Then, an optimum support-reducing decomposition can be found among $\binom{n}{s}$ different decompositions.

Algorithm 4.1: (Exhaustive Decomposition)
Given an index generation function f, and s, the number of bound variables, this algorithm returns a bound set that produces the **minimum column multiplicity** among $\binom{n}{s}$ bound sets.

1) /*** Obtain an initial solution ***/
   $X_{min} \leftarrow \{x_{n-s+1}, \ldots, x_{n-1}, x_n\}$,
   $\mu_{min} \leftarrow \mu(f : X_{min})$,and
   $r_{min} \leftarrow rails(f : X_{min})$.
2) Call **backtrack_r**$(\phi, 1)$.
   Procedure **backtrack_r**(X_1, v)
   a) If $|X_1| = s$, then
      i) If $\mu(f : X_1) < \mu_{min}$, then
         $X_{min} \leftarrow X_1$, $\mu_{min} \leftarrow \mu(f : X_{min})$, and
         $r_{min} \leftarrow rails(f : X_{min})$.
      ii) return.
   b) If $n - v + 1 < s - |X_1|$, then return.
   c) For all possible i such that $v \leq i \leq n$, call **backtrack_r**$(X_1 \cup \{x_i\}, i+1)$.
3) X_{min} gives an optimum bound set,
   μ_{min} gives the minimum column multiplicity, and
   r_{min} gives the minimum number of rails.

(Explanation) A recursive procedure **backtrack_r** generates all possible bound sets with s elements, and checks their column multiplicities. It has two arguments: X_1, the bound set, and $v (0 \leq v \leq n+1)$, an integer showing the index of the variable. In the call of $|X_1| = s$, the procedure checks if X_1 is a better solution or not. If so, then it updates the best solution. In step b), it prunes the recursive calls that never produces a bound set. In the call of $|X_1| \neq s$, the procedure add a variable with an index greater than or equal to v, and produces a bound set with $|X_1|+1$ elements. The subroutine $rails(f : X_1)$ obtains the number of rails when the bound set is X_1.

B. Fast Method

Index generation functions have a special property that is useful to find optimum support-reducing decompositions efficiently.

Theorem 4.1: Let $f(X)$ be an index generation function. If $X_A \subseteq X_B \subseteq X$, then $\mu(f : X_A) \leq \mu(f : X_B)$.

(Proof) In a decomposition chart of an index generation function, the column multiplicity is equal to the number of columns with non-zero elements + 1 (if there exists a column with all 0's). Consider two decomposition charts: The first chart has the bound set X_A, while the second chart has the bound set $X_A \cup x_i$. Note that the second chart has one more bound variables than the first chart. Thus, the number of columns in the second chart is twice of that in the first chart. In this case, the column multiplicity of the second decomposition chart is not less than that of the the first chart, since the number of columns with non-zero element(s) never decrease with the increase of the bound variables. From this, we have the theorem. □

Thus, the column multiplicity increases monotonically with the increase of the number of bound variables.

Example 4.1: Consider the decomposition chart in Fig. 3.1. In this case, $X_1 = \{x_1, x_2, x_4\}$ denotes the bound variables, while $X_2 = \{x_3, x_5\}$ denotes the free variables, and the column multiplicity is four. When x_3 is moved to the bound set, the column multiplicity will be seven. On the other hand, when x_5 is moved to the bound set, the column multiplicity will be five. In both cases, the column multiplicity increase with a move of a free variable to the bound set. ■

With Theorem 4.1, we can drastically reduce the search space for the decompositions of index generation functions. Note that in an ordinary logic function, Theorem 4.1 does not hold as shown in Example 4.2.

Example 4.2: Consider the decomposition chart in Fig. 2.1. In this case, $X_1 = \{x_1, x_2\}$ denotes the bound variables, while $X_2 = \{x_3, x_4\}$ denotes the free variables, and the column multiplicity is four.

When x_3 is moved to the bound set, the column multiplicity will be three. Also, when x_4 is moved to the bound set, the column multiplicity will be three. In both cases, the column multiplicity decrease with a move of a free variable to the bound set. ■

Thus, in the case of logic function, the column multiplicity can be reduced with a move of free variable to bound set.

Assume that we have a decomposition that produces a circuit with r rails. Then, our objective is to find a decomposition with fewer rails than r. In this case, if a decomposition $f(X_1, X_2)$ has a column multiplicity $\mu > 2^{r-1}$, then any decomposition $f(X_3, X_4)$, where $X_3 \supset X_1$ need not be considered.

Algorithm 4.2: (Fast Decomposition)

Given an index generation function f, and s, the number of bound variables, this algorithm returns a bound set that produces the **minimum rails** among $\binom{n}{s}$ bound sets.

1) /*** Obtain an initial solution ***/
   $X_{min} \leftarrow \{x_{n-s+1}, \ldots, x_{n-1}, x_n\}$,
   $\mu_{min} \leftarrow \mu(f : X_{min})$, and
   $r_{min} \leftarrow rails(f : X_{min})$.
2) Call $\mathbf{backtrack_r}(\phi, 1)$.
   Procedure $\mathbf{backtrack_r}(X_1, v)$
   a) If $|X_1| = s$, then
      i) If $\mu(f : X_1) < \mu_{min}$, then
         $X_{min} \leftarrow X_1$, $\mu_{min} \leftarrow \mu(f : X_{min})$, and
         $r_{min} \leftarrow rails(f : X_{min})$.
      ii) return.
   b) If $n - v + 1 < s - |X_1|$, then return.
   c) /*** Prune the search tree ***/
      If $|X_1| \geq r_{min}$ and $rails(f : X_1) \geq r_{min}$, then return.
   d) For all possible i such that $v \leq i \leq n$, call $\mathbf{backtrack_r}(X_1 \cup \{x_i\}, i + 1)$.
3) X_{min} gives an optimum bound set;
   μ_{min} gives the column multiplicity that produces the minimum rails; and
   r_{min} gives the minimum number of rails.

(Explanation) Algorithm 4.2 is similar to Algorithm 4.1. The only difference is step c), that is appended to Algorithm 4.2. If the number of the bound variables is equal to or greater than r_{min}, then the number of rails is computed, and if it is equal to or greater than r_{min}, then skip the search. This algorithm produces the bound set with the minimum rails, but the generated column multiplicity is not necessary the minimum.

Example 4.3: Consider the function in Example 3.1.

1) In this function, $n = 5$ and $k = 7$. Assume that $s = 4$.
2) Let the initial solution be $X_{min} = \{x_2, x_3, x_4, x_5\}$. Since $\mu(f : X_{min}) = 8$ and $rails(f : X_{min}) = 3$, we have $\mu_{min} = 8$ and $r_{min} = 3$.
3) Fig.4.1 shows the tree to find the bound set X_1. Each node corresponds to a selection of a variables. This figure shows that, for the first variable, either x_1 or x_2 can be selected. When x_1 is selected for the first variable, for the second variable, either x_2 or x_3 can be selected. When x_2 is selected for the first variable, for other variables, x_3, x_4 or x_5 can be selected. The depth-first search will be performed in this tree. When the number of elements in X_1 is equal to or greater than r_{min}, we skip the search.
4) In the beginning, let $X_1 = \phi$. By adding variables to X_1 until $|X_1| \geq r_{min}$, we have $X_1 = \{x_1, x_2, x_3\}$. In this decomposition, since $\mu(f : X_1) = 7$ and $r = 3 \geq r_{min}$, we skip the search.
5) By selecting other variables, we have $X_1 = \{x_1, x_2, x_4\}$. Since $\mu(f : X_1) = 4$ and $r = 2 < r_{min}$, we may add more variables to find a better decomposition. By appending x_5 to X_1, we have $X_1 = \{x_1, x_2, x_4, x_5\}$. Since $|X_1| = s$, we compute the column multiplicity. Since $\mu(f : X_1) = 5 < \mu_{min}$ and $r = 3$, we update the solution as follows: $X_{min} = \{x_1, x_2, x_4, x_5\}$, $\mu_{min} = 5$, and $r_{min} = 3$.
6) By changing the bound variables, we have $X_1 = \{x_1, x_3, x_4\}$. Since $\mu(f : X_1) = 7$, $r = 3 \geq r_{min}$, we skip this combination.
7) Similarly, $X_1 = \{x_2, x_3, x_4\}$ is checked, but it is skipped.
8) Here, we terminate the search. $X_{min} = \{x_1, x_2, x_4, x_5\}$ is an optimum bound set, and $\mu_{min} = 5$ and $r_{min} = 3$.

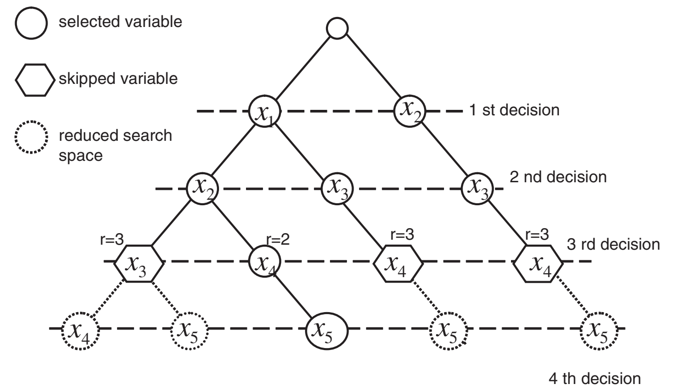

Fig. 4.1. Tree to Find the Bound Set.

C. Optimum Decomposition

To find the decomposition that minimizes the total memory size in Fig. 1.1, we have to find the optimum decompositions for $s \in \{2, 3, \ldots, n-2\}$, and compute $Size = r2^s + q2^{n-s+r}$.

TABLE 5.1
DECOMPOSITION OF IP ADDRESS TABLE.

Name	n	k	s	Exhaustive μ	r	CPU min	Fast μ	r	CPU sec
$Ip1670$	32	1670	21	1277	11	866	1669	11	18.32
$Ip3288$	32	3288	22	2669	12	1479	3288	12	29.40
$Ip4591$	32	4591	22	3819	12	1710	3902	12	8.48
$Ip7903$	32	7903	22	5169	13	2908	7898	13	58.53

However, since $1 \leq r \leq q$, we can reduce the search space. For example, consider the case of $n = 60$ and $k = 1000$. Since $q = 10$, the search space is reduced to $26 \leq s \leq 39$. If we solve the case for $s = 35$ first, and obtain the result $r = 10$, then the remaining search space is reduced to $26 \leq s \leq 34$.

V. EXPERIMENTAL RESULTS

We implemented Algorithms 4.1 and 4.2 for a workstation using 3.4 GHz Intel Xeon E3-1270 v3 3.5GHz 8 Core processor, and 32 Giga bytes of memory, on the Ubuntu 14.04 (64-bit) operating system.

In the program, an index generation function is represented by an array of integers. To compute the column multiplicity, a table method is used. A straightforward method is used to make composition functions. No *don't care* optimization technique [15] nor encoding technique [10] were used to simplify functions.

A. IP Address Tables

We collected distinct IP addresses of computers that accessed our web site over a period of one month. We considered four lists of different values of k. The original numbers of variables are 32 for all the functions. Table 5.1 shows the results. The first column shows the name of the function. The second column shows the number of the variables: n. The third column shows the number of registered vectors: k. The fourth column shows the number of bound variables: s. The middle three columns headed "Exhaustive" show the results obtained by Algorithm 4.1. The fifth column shows the column multiplicity: μ. The sixth column shows the number of rails : r. The seventh column shows the CPU time in minutes. The last three columns headed "Fast" show the result obtained by Algorithm 4.2. μ and r denote the same things as the middle columns. The last column denotes the CPU time in seconds. As shown in Table 5.1, Algorithm 4.2 (Fast) is much faster than Algorithm 4.1 (Exhaustive). Note that for each function, the number of rails r is considerably smaller than that of the bound variables s. Thus, the functions have efficient support-reducing decompositions. The number of the bound variables s is chosen so that the total memory size is minimized, when the function is implemented by two modules. To find such s, we used the technique in IV-C. In many cases, $s = \lfloor \frac{n+\log_2(k+1)}{2} \rfloor$.

B. Lists of English Words

We made three lists of English words: *Dic1730*, *Dic3366*, and *Dic4705*. 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 characters, we append blanks to make the length of words 8. We represent each alphabetic character by 5 bits. So, in the lists, all the words are represented by 40 bits. The numbers of words in the lists are 1730, 3366, and 4705, respectively. Each English word in a list has a unique index, an integer from 1 to k, where $k = 1730$ or 3366 or 4705. Table 5.2 shows the results of decomposition. For these functions, $r < \lceil \log_2(k+1) \rceil$, i.e., the column multiplicities were greatly reduced.

Again, these functions have efficient support-reducing decompositions. For these functions, computation did not stop in an allocated time (2880 minutes) for exhaustive search, which are shown in "-" in the table.

C. Lists of English Words Having the Same Word Length

Nine additional lists of English words were derived in a similar way. However, in this case, each list consist of words with the same length. Thus, the different list have different number of characters (variables). Also, no blank character(s) are contained in the lists.

Table 5.3 shows the results. For example, *Cha04* shows the list of English words consisting of exactly four characters. Thus, it has $n = 5 \times 4 = 20$ input variables. Again, these functions have efficient support-reducing decompositions.

For $Cha12$, since $n = 60$ and $s = 33$, there exist $\binom{60}{33} \simeq 8.8 \times 10^{16}$ combinations to consider. Thus, an exhaustive method is impractical for this problem. On the other hand, Algorithm 4.2 successfully obtained an optimum decomposition.

D. Random Functions

We generated nine random index generation functions that have the same values of (n, k), as the previous experiments. Table 5.4 shows the results.

Unlike non-random functions, column multiplicities of random functions were hard to reduce, and $\mu \simeq k + 1$. Thus, for all the random functions, $r = \lceil \log_2(k+1) \rceil$. Again, in

TABLE 5.2
DECOMPOSITION OF ENGLISH WORD LISTS

Name	n	k	s	Exhaustive μ	r	CPU min	Fast μ	r	CPU sec
$Dic1730$	40	1730	25	–	–	–	850	10	10.59
$Dic3366$	40	3366	26	–	–	–	1264	11	32.93
$Dic4705$	40	4705	26	–	–	–	2458	12	410.84

"-" denotes timeout.

TABLE 5.3
DECOMPOSITION OF ENGLISH WORD LISTS

Name	n	k	s	Exhaustive μ	r	CPU min	Fast μ	r	CPU sec
$Cha04$	20	768	15	378	9	a	378	9	0.00
$Cha05$	25	820	17	567	10	b	586	10	0.38
$Cha06$	30	809	20	626	10	75	626	10	1.38
$Cha07$	35	701	22	–	–	–	583	10	30.92
$Cha08$	40	548	25	–	–	–	474	9	1.93
$Cha09$	45	378	27	–	–	–	333	9	35.55
$Cha10$	50	271	29	–	–	–	232	8	3.74
$Cha11$	55	142	31	–	–	–	126	7	1.49
$Cha12$	60	74	33	–	–	–	64	6	3097.94

a: 60 milli seconds, b:17.27 seconds, "-": timeout.

TABLE 5.4
DECOMPOSITION OF RANDOM INDEX GENERATION FUNCTIONS

				Exhaustive			Fast		
Name	n	k	s	μ	r	CPU min	μ	r	CPU sec
$32i1670$	32	1670	21	1622	11	884	1670	11	0.69
$32i3288$	32	3288	22	3277	12	1740	3289	12	1.41
$32i4591$	32	4591	23	4577	13	1901	4592	13	39.27
$32i7903$	32	7903	22	7877	13	2663	7898	13	5.70
$20i0768$	20	768	15	746	10	c	763	10	0.00
$25i0820$	25	820	17	808	10	d	821	10	0.02
$30i0809$	30	809	20	804	10	76	810	10	0.09
$35i0701$	35	701	22	–	–	–	702	10	1.48
$40i0548$	40	548	25	–	–	–	549	10	38.84
$45i0378$	45	378	27	–	–	–	379	9	1.35
$50i0271$	50	271	29	–	–	–	272	9	228.10
$55i0142$	55	142	31	–	–	–	143	8	35.48
$60i0074$	60	74	33	–	–	–	75	7	8.64

c: 60 milli seconds, d:17.40 seconds,
"-": timeout.

the case of random functions, we obtained support-reducing decompositions.

E. Computation Time

To estimate the computation time for the decomposition programs, we randomly generated index generation functions of n variables with weight $k = 1000$. For each n, we generated 10 functions, and the number of bound variables is set to $s = n/2$. The computation time is approximated by $\alpha\beta^n + \gamma$, and their coefficients were obtained by a regression analysis. For the exhaustive method, we had $\alpha = 2.060 \times 10^{-4}$, $\beta = 2.012$, and $\gamma = 1.007$, while for the fast method, we had $\alpha = 2.910 \times 10^2$, $\beta = 1.167$, and $\gamma = 1.016$.

VI. CONCLUSION AND COMMENTS

A. Conclusion

In this paper, we presented an exact method to find an optimum support-reducing decomposition for index generation functions. The algorithm uses a special property of index generation functions to reduce the search space. Experimental results show that our algorithm obtained an exact optimum decomposition for a function with $n = 60$ inputs, where the number of the bound variables is $s = 33$. Note that an exhaustive method failed to find a solution, since it requires to check $\binom{60}{33} \simeq 8.8 \times 10^{16}$ different decompositions.

B. Comments

The computing time increases exponentially with n. Thus, although the presented method is efficient, it works for the problems with limited size of n. Computation time heavily depends on the property of the function. For example, when $n = 60$, $k = 74$ and $s = 33$, the English word list took 3098 seconds, while a randomly generated function took only 8.64 seconds.

Fortunately, we have a good heuristic algorithm [14]. Our experiments shows that for the benchmark functions in the paper, the solutions obtained by the heuristic method are proved to be optimum by Algorithm 4.2, except for $Cha12$. $Char12$ was one of the most time-consuming problems. The heuristic method took less than 1 milli second to obtain a 7-rail solution, while the exact method took 3098 seconds to obtain a 6-rail solution.

In the experiment, we selected the value of s so that the cost of the circuit is minimized when the function is implemented by two LUTs. In practice, the function can also be implemented by three or more LUTs. In such a case, the algorithm can be used iteratively to reduce the total size.

C. Limitation of the Method

Moderns routers use ternary content addressable memories (TCAMs) which allow for *don't cares* in the input pattern. Unfortunately, the presented method is not applicable for such cases. To implement TCAM functions, we need *prefix expansion* [8], and multiple IGUs [12].

ACKNOWLEDGEMENTS

This research is supported in part by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS).

REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions," *Inter. Symp. on the Theory of Switching*, pp. 74-116, April 1957.

[2] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions,"*ICCAD-97*, pp. 78-82, Nov. 1997.

[3] H. A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Co., Princeton, NJ, 1962.

[4] V. Kravets and K. Sakallah, "Constructive library-aware synthesis using symmetries,"*Proc. DATE-2000*, pp. 208-213.

[5] Y. Matsunaga, "An exact and efficient algorithm for disjunctive decomposition,"*SASIMI'98*, pp. 44-50, Oct. 1998.

[6] D. P. Mehta, and S. Shani, *Handbook of Data Structure and Applications*, Chapman and Hall/CRC, Oct. 2004.

[7] S-I. Minato and G. De Micheli, "Finding all simple disjunctive decompositions using irredundant sum-of-products forms," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 111-117, Oct. 1998.

[8] H. Nakahara, T. Sasao, M. Matsuura, H. Iwamoto, and Y. Terao,"A memory-based IPv6 lookup architecture using parallel index generation units," *IEICE Trans. Inf. and Syst.* Vol. E98-D, No. 2, pp. 262-271, Feb., 2015.

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

[10] T. Sasao, *Memory-Based Logic Synthesis*, Springer, 2011.

[11] T. Sasao, "Index generation functions: Tutorial,"*Journal of Multiple-Valued Logic and Soft Computing*, Vol. 23, No. 3-4, pp. 235-263, 2014.

[12] T. Sasao, "A realization of index generation functions using multiple IGUs," *Inter. Symp. on Multiple-Valued Logic*, (ISMVL-2016), Sapporo, Japan, May 17-19, 2016. pp. 113-116.

[13] T. Sasao and J. T. Butler, "Decomposition of index generation functions using a Monte Carlo method," *Inter. Symp. on Logic and Synthesis*, (IWLS-2016), Austin, TX, U.S.A, June 10-11, 2016.

[14] T. Sasao, K. Matsuura, Y. Iguchi,"A heuristic decomposition of index generation functions with many variables,"*The 20th Workshop on Synthesis And System Integration of Mixed Information Technologies* (SASIMI-2016), Kyoto, Oct. 24, 2016.

[15] W. Wan and M. A. Perkowski, "A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping," *European Design Automation Conference* (EURO-DAC '92), pp.230-235, 1992.