2018 IEEE 48th International Symposium on Multiple-Valued Logic

An Exact Optimization Method Using ZDDs for Linear Decomposition of Index Generation Functions

Shinobu Nagayama* Tsutomu Sasao† Jon T. Butler‡

*Dept. of Computer and Network Eng., Hiroshima City University, Hiroshima, JAPAN
†Dept. of Computer Science, Meiji University, Kawasaki, JAPAN
‡Dept. of Electr. and Comp. Eng., Naval Postgraduate School, Monterey, CA USA

***Abstract*—This paper proposes an exact optimization method using zero-suppressed binary decision diagrams (ZDDs) for linear decomposition of index generation functions. The proposed method searches for an exact optimum solution by recursively dividing an index set of an index generation function. Since ZDDs can represent sets compactly and uniquely, they can also represent partitions of an index set compactly and uniquely. Thus, the proposed method can reuse partial solutions (partitions of an index set) efficiently by using ZDDs, and avoid redundant solution search. Experimental results using benchmark index generation functions show the effectiveness of ZDDs.**

***Index Terms*—Index generation functions; linear decomposition; zero-suppressed binary decision diagrams; logic design; exact optimization method.**

I. Introduction

Index generation functions [6], [7] are logical models of index searches. Since index searches are widely used as a basic operation in many applications, such as detection of computer viruses and packet classification, fast hardware for index generation functions has been required. In addition to processing speed, an ability for quick updating of index generation functions is also important particularly in these network applications, because virus patterns and rules for classification are frequently updated. Thus, there is merit to a memory-based hardware design.

An efficient memory-based hardware design method for index generation functions, one using *linear decomposition* [1], [5] has been proposed [9]. In this design method, an index generation function $f(x_1, x_2, \ldots, x_n)$ is decomposed into two parts: L and G, as shown in Fig. 1. The first part L realizes linear functions y_i $(i = 1, 2, \ldots, p)$ in a linear decomposition of f. L is realized by a programmable architecture [9] with EXOR gates, registers, and multiplexers. The second one G realizes a remaining function (general function) in a linear decomposition of f. G is realized by a $(2^p \times q)$-bit memory, where p is the number of linear functions, and q is the number of bits needed to represent function values of f.

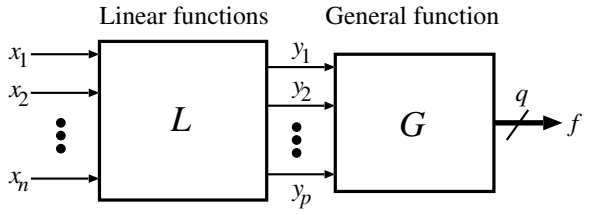

Fig. 1. Linear decomposition of an index generation function [9].

Since memory size of this hardware strongly depends on the number of linear functions, p, minimization of p is indispensable to obtain a practical implementation. Thus, various minimization methods have been proposed [3], [4], [8], [9], [11], [12], [13], [14]. Most of them are heuristic methods, and as far as we know, only a few exact minimization methods [4], [13], [14] have been proposed. However, devising an efficient exact minimization method is not only academically but also practically significant, because it becomes a basis for evaluating the quality of heuristic methods.

The exact minimization methods proposed in [13], [14] are based on a SAT solver, and solve the linear decomposition problem by reducing it to a SAT problem. Thus, they have little room for improvement unless a SAT solver is improved. On the other hand, the method proposed in [4] is an emerging dedicated method for the linear decomposition problem, and thus, it still has enough room for improvement. Hence, in this paper, we focus on improvement of the dedicated method, and propose an exact optimization method for linear decomposition of index generation functions.

The proposed method uses zero-suppressed binary decision diagrams (ZDDs) to represent partial solutions of the problem compactly and uniquely. By using ZDDs, the proposed method can reuse partial solutions efficiently to avoid redundant solution search that the existing method [4] does not avoid.

The rest of this paper is organized as follows: Section II defines index generation functions and linear decomposition. Section III formulates the minimization problem for the number of linear functions, provides a brief overview of the existing exact minimization method [4], and proposes an exact minimization method using ZDDs. Section IV shows experimental results using some benchmark index generation functions, and Section V concludes the paper.

2378-2226/18/$31.00 ©2018 IEEE
DOI 10.1109/ISMVL.2018.00033

TABLE I
EXAMPLE OF INDEX GENERATION FUNCTION [4].

Registered vectors				indices
x_1	x_2	x_3	x_4	f
0	0	0	1	1
0	0	1	0	2
0	1	0	0	3
1	1	0	1	4

TABLE II
GENERAL FUNCTIONS g_1 AND g_2 IN LINEAR DECOMPOSITION OF f [4].

y_1	y_2	g_1	g_2
0	0	1	2
0	1	2	1
1	0	3	3
1	1	4	4

II. PRELIMINARIES

We briefly define index generation functions [6], [7] and their linear decompositions [1], [5], [9].

Definition 1: An **incompletely specified index generation function**, or simply **index generation function**, $f(x_1,x_2,\ldots,x_n)$ is a multi-valued function, where k assignments of values to binary variables $x_1,x_2,\ldots,x_n$ map to $K=\{1,2,\ldots,k\}$. That is, the variables of f are binary-valued, while f is k-valued. Further, there is a one-to-one relationship between the k assignments of values to $x_1,x_2,\ldots,x_n$ and K. Other assignments are left unspecified. The k assignments of values to $x_1,x_2,\ldots,x_n$ are called the set of **registered vectors**. K is called the set of **indices**. $k=|K|$ is called the **weight** of the index generation function f.

Example 1: Table I shows a 4-variable index generation function with weight four. Note that, in this function, input values other than 0001, 0010, 0100, and 1101 are NOT assigned to any function values. □

Definition 2: Let $K=\{1,2,\ldots,k\}$ be a set of indices of an index generation function. If $K=S_1\cup S_2\cup\ldots\cup S_u$, each $S_i\neq\emptyset$, and $S_i\cap S_j=\emptyset$ $(i\neq j)$, then $\mathcal{P}=\{S_1,S_2,\ldots,S_u\}$ is a **partition** of the set of indices K. When all the subsets S_i are singletons (i.e., $|S_i|=1$), $|\mathcal{P}|=|K|=k$.

An arbitrary n-variable index generation function with weight k can be realized by a $(2^n\times q)$-bit memory, where $q=\lceil\log_2(k+1)\rceil$. To reduce the memory size, linear decomposition is effective [9].

Definition 3: **Linear decomposition** of an index generation function $f(x_1,x_2,\ldots,x_n)$ is a representation of f using a general function $g(y_1,y_2,\ldots,y_p)$ and linear functions y_i:

$$y_i(x_1,x_2,\ldots,x_n)=a_{i1}x_1\oplus a_{i2}x_2\oplus\ldots\oplus a_{in}x_n \quad (i=1,2,\ldots,p),$$

where $a_{ij}\in\{0,1\}$ $(j=1,2,\ldots,n)$, and, for all registered vectors of the index generation function, the following holds:

$$f(x_1,x_2,\ldots,x_n)=g(y_1,y_2,\ldots,y_p).$$

Each y_i is called a **compound variable**. For each y_i, $\sum_{j=1}^{n}a_{ij}$ is called a **compound degree** of y_i, denoted by $deg(y_i)$, where a_{ij} is viewed as an integer, and $\sum$ as an integer sum.

Definition 4: An inverse function of a general function $z=g(y_1,y_2,\ldots,y_p)$ in a linear decomposition is a mapping from $K=\{1,2,\ldots,k\}$ to a set of p-bit vectors $\{0,1\}^p$, denoted by $g^{-1}(z)$. In this inverse function $g^{-1}(z)$, a mapping obtained by focusing only on the i-th bit of the p-bit vectors: $K\rightarrow\{0,1\}$ is called an **inverse function to a compound variable** y_i, denoted by $(g^{-1})_i(z)$.

Definition 5: Let $ON(y_i)=\{z \mid z\in K,(g^{-1})_i(z)=1\}$, where $K=\{1,2,\ldots,k\}$ and $(g^{-1})_i(z)$ is an inverse function of $g(y_1,y_2,\ldots,y_n)$ to y_i. $|ON(y_i)|$ is called the **cardinality of** y_i or informally the **number of 1s included in** y_i.

Example 2: The index generation function f in Example 1 can be represented by $y_1=x_2$, $y_2=x_1\oplus x_3$, and $g_1(y_1,y_2)$ shown in Table II. In this case, $deg(y_1)=1$ and $deg(y_2)=2$, respectively. f can be also represented by $y_1=x_2$, $y_2=x_4$, and $g_2(y_1,y_2)$ in the same table. In this case, both $deg(y_1)$ and $deg(y_2)$ are 1. In either case, f can be realized by the architecture in Fig. 1 with a $(2^2\times 3)$-bit memory.

For $g_2(y_1,y_2)$ in Table II, its inverse functions to y_1 and y_2 are $(g_2^{-1})_1(z)$ and $(g_2^{-1})_2(z)$, respectively. We have $(g_2^{-1})_1(2)=0$, $(g_2^{-1})_1(1)=0$, $(g_2^{-1})_1(3)=1$, and $(g_2^{-1})_1(4)=1$. Similarly, $(g_2^{-1})_2(2)=0$, $(g_2^{-1})_2(1)=1$, $(g_2^{-1})_2(3)=0$, and $(g_2^{-1})_2(4)=1$. The cardinalities of both y_1 and y_2 are 2. □

In this way, by using linear decomposition, memory size needed to realize an index generation function can be reduced significantly. But, to realize a compound variable with compound degree d, $(d-1)$ 2-input EXOR gates are required. Thus, a lower compound degree is desirable when the memory size is equal.

III. OPTIMIZATION OF LINEAR DECOMPOSITION

This section formulates the minimization problem of the number of linear functions, and shows exact minimization methods to solve the problem.

A. Formulation of Minimization Problem

Since the architecture in Fig. 1 realizes an index generation function with EXOR gates, registers, multiplexers, and a $(2^p\times q)$-bit memory, to obtain an optimum realization of an index generation function, we have to solve the following problem:

Problem 1: Given an index generation function f and an integer t, find a linear decomposition of f such that the number of linear functions p is minimum, and compound degrees are at most t.

The constraint on compound degrees t is given to constrain not only solution space, but also delay and area of the circuit L realizing the linear functions.

Example 3: For linear decompositions of f in Example 2, the decomposition with $y_1=x_2$, $y_2=x_4$, and $g_2(y_1,y_2)$ is optimum when $t=1$. □

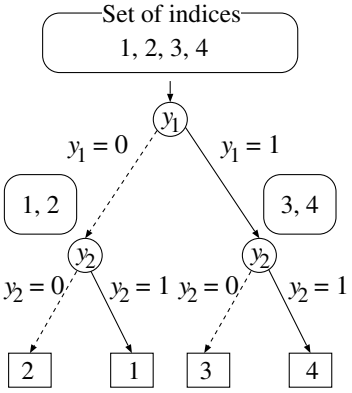

Fig. 2. Binary decision tree for g_2 of Table II.

Algorithm 1: Overview of the existing method [4]

Input: an index generation function with weight k and an upper bound t on compound degrees Output: a set of compound variables and its size h_{min}

Let $\mathcal{P} = \{K\}, h = 0$, and iterate the following recursively.
min_search($\mathcal{P}, h$) {
  if ($\|\mathcal{P}\| = k$) { **update_solution**(h); return; }
  if (**bound_condition**($\mathcal{P}, h$) is satisfied) return;
  branch($\mathcal{P}$, t, h);
}

B. Existing Method Based on Partition of Indices

To provide a basis of comparison between the proposed method and the existing one [4], we give a brief review below.

Problem 1 can be considered as a problem of minimizing the height of a binary decision tree constructed by compound variables [3].

Example 4: Fig. 2 shows a binary decision tree of the smallest height that divides the set of indices into singletons by compound variables y_1 and y_2. This corresponds to g_2 in Table II. □

The existing method finds a binary decision tree with the smallest height by a branch and bound algorithm. It constructs binary decision trees in a top-down manner while dividing a given set of indices recursively by selecting a compound variable one by one, and finds the best one by comparing heights of the trees. Algorithm 1 shows the overview of the existing method.

Algorithm 1 searches for a solution recursively while constructing a binary decision tree with height h. When $|\mathcal{P}| = k$ (i.e., the set of indices is divided into singletons), a solution (a set of h compound variables) is obtained. The procedure *update_solution*() compares the obtained solution with the current solution, and updates the current solution if the obtained one is better. The procedure *branch*() explores the solution space by selecting a compound variable using two cost functions proposed for the heuristic method [3]:

$$cost_1(\mathcal{P}, y_i) = \sqrt{\sum_{S \in \mathcal{P}} \left(\frac{|S|}{2} - |S \cap ON(y_i)| \right)^2}$$

and

$$cost_2(\mathcal{P}, y_i) = \max_{S \in \mathcal{P}} \{ \max \{ |S \cap ON(y_i)|, |S \setminus ON(y_i)| \} \},$$

where $\mathcal{P}$ is a partition of a set of indices with already selected compound variables. The procedure *bound_condition*() detects an ineffective solution using the lower bound discussed below in Theorem 1, and prunes it.

Theorem 1: [4] Let m be the number of indices in a set, and c be the number of 1s in compound variables. When $c < \frac{m}{2}$, at least

$$lower(m, c) = \left\lfloor \frac{m}{c} \right\rfloor + \lceil \log_2(c) \rceil - 1$$

compound variables are needed to divide the set into m singletons.

C. Proposed Improvement Method

Although the existing method [4] is promising, it can be improved further. To search for the optimum solution more efficiently, we propose two improvements described in the following subsections.

1) Improvement Using ZDDs: Before describing an improvement using ZDDs, we briefly define ZDDs.

Definition 6: A ***zero-suppressed binary decision diagram (ZDD)*** [2] is a rooted directed acyclic graph (DAG) representing a logic function. It consists of two terminal nodes representing function values 0 and 1, and nonterminal nodes representing input variables. Each nonterminal node has two outgoing edges, 0-edge and 1-edge, that correspond to the values of the input variables. Neither terminal node has outgoing edges.

A ZDD is obtained by repeatedly applying the Shannon expansion $f = \overline{x_i} f_0 \vee x_i f_1$ to a logic function, where $f_0 = f(0 \rightarrow x_i)$, and $f_1 = f(1 \rightarrow x_i)$, and by applying the following two reduction rules:

1) Coalesce equivalent sub-graphs.
2) Delete nonterminal nodes whose 1-edge points to the terminal node representing 0, and redirect edges that point to the deleted node, to the node, to which the 0-edge of the deleted node has pointed.

As is well known, ZDDs can represent sets of combinations compactly and uniquely [2]. A partition of an index set $\mathcal{P} = \{S_1, S_2, \ldots, S_u\}$ can be also represented compactly and uniquely using a ZDD, as shown in the example below.

Example 5: Let an index set be $K = \{1,2,3,4,5,6\}$, and a partition of K be $\mathcal{P} = \{\{1,3,6\},\{2,5\},\{4\}\}$. Fig. 3 shows a ZDD for $\mathcal{P}$. In Fig. 3, dashed lines and solid lines denote 0-edges and 1-edges, respectively. The number of nonterminal nodes is 6. □

Theorem 2: Let an index set be $K = \{1,2,\ldots,k\}$. For any partition $\mathcal{P}$ of K, the number of nonterminal nodes in a ZDD for $\mathcal{P}$ is k, regardless of the variable order.

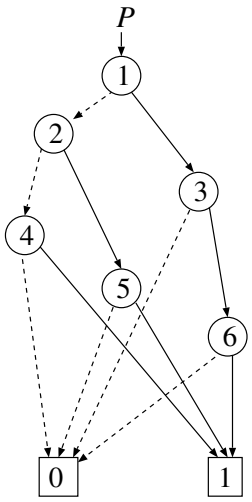

Fig. 3. ZDD for $\mathcal{P} = \{\{1,3,6\},\{2,5\},\{4\}\}$.

Proof: Let a 1**-path** in a ZDD be a sequence of edges and nodes leading from the root node to the terminal node representing 1. A 1-path in a ZDD for $\mathcal{P}$ represents a subset S_i in $\mathcal{P}$, and a pair of a 1-edge and its nonterminal node on the 1-path represents an index in the subset S_i. Thus, the number of 1-paths is exactly $|\mathcal{P}|$, and the number of pairs of a 1-edge and its nonterminal node is exactly $|S_i|$. Since from Definition 2, $S_i \cap S_j = \emptyset$ $(\forall S_i, S_j \in \mathcal{P})$, more than one 1-path do not share a pair of a 1-edge and its nonterminal node. Thus, the total number of pairs of a 1-edge and its nonterminal node on all 1-paths is exactly k, regardless of the order of the pairs.

If the number of pairs of a 1-edge and its nonterminal node in a ZDD were larger than k, the ZDD would represent a subset of indices that does not exist in $\mathcal{P}$. That is, when all 1-paths are removed from a ZDD for $\mathcal{P}$, remaining edges are only 0-edges due to the zero-suppressed reduction rule for ZDDs. Therefore, the number of nonterminal nodes in a ZDD for $\mathcal{P}$ is k, regardless of its variable order. ∎

As described in Section III-B, the method based on a partition of indices searches for the optimum solution while dividing a given index set repeatedly. Thus, the same partition of indices tends to repeatedly appear during solution search. However, once the minimum number of compound variables needed to divide a partition of indices into singletons is found, we do not need to search for the minimum number of compound variables for the same partition again, and the obtained subsolution can be reused to prune the search space. The question is how to represent partitions of indices compactly and uniquely. To answer this question, we use ZDDs to represent partitions of indices.

Algorithm 2 shows the overview of the improved method using ZDDs. A ZDD is constructed for each partition using *Change* and *Union* operations that are basic operations in a ZDD package [2]. Then, the ZDD is checked to determine whether the current subsolution has already been obtained.

Algorithm 2: Overview of the improved method using ZDDs

Input: an index generation function with weight k and an upper bound t on compound degrees
Output: a set of compound variables and its size h_{min}

Let $\mathcal{P} = \{K\}, h = 0$, and iterate the following recursively.
min_search($\mathcal{P}, h$) {
  if ($|\mathcal{P}| = k$) { update_solution(h); return; }
  Construct a ZDD for $\mathcal{P}$;
  Search for ZDD_subsolution in ZDD_History();
  if ((**ZDD_subsolution is found**) and
    (its min_lower + $h \geq h_{min}$)) return;
  if (bound_condition($\mathcal{P}, h$) is satisfied) return;
  min_lower = branch($\mathcal{P}$, t, h);
  Add the ZDD and min_lower to ZDD_History;
}

This checking is made by searching for the equivalent one in the history of ZDDs. Such a search (equivalence checking) is what ZDDs can do best; it is done in $O(1)$ time. If the subsolution has already been obtained, and it does not improve the current solution, then the solution search is pruned. On the other hand, when any subsolution has not been obtained yet, the solution search is performed, and after that, the ZDD is stored to the history with the obtained subsolution. In this way, redundant solution search is efficiently pruned using ZDDs.

2) Improvement by Commutative Law of EXOR: The procedure *branch*() in the existing method selects the same compound variable repeatedly since the commutative law of EXOR operations is not considered when a compound variable is produced. From Definition 3, compound variables

$$y(x_1, x_2, \ldots, x_n) = a_1x_1 \oplus a_2x_2 \oplus \ldots \oplus a_nx_n \quad a_i \in \{0,1\}\ (i = 1,2,\ldots,n)$$

depend not on permutations of x_i but on combinations of x_i due to the commutative law of EXOR operations. To produce compound variables by combinations of x_i, we use a table indicating whether x_i has been already compounded for each compound variable (i.e. in each branch).

The size of each table is n, and at most n compound variables are produced in a search. Thus, additional memory size for this improvement is $O(n^2)$. Since this is almost the same memory size needed to store registered vectors, this improvement is scalable. It reduces search space significantly, as will be shown in experimental results.

IV. Experimental Results

The proposed exact minimization methods are implemented in the C language, and run on the following computer environment: CPU: Intel Core2 Quad Q6600 2.4GHz, memory: 4GB, OS: CentOS 5.7 Linux, and C-compiler: gcc -O2 (version 4.1.2).

A. On Reduction of Search Space

To evaluate the effectiveness of the proposed improvement methods, we compare search space size for the following three methods:

TABLE III
COMPARISON OF METHODS IN TERMS OF SEARCH SPACE.

Benchmark functions	Compound degrees	h_{min}	Existing [4]	Comm. law	ZDD
1-out-of-10	$t=1$	9	9	9	9
	$t=2$	6	1,975,364	135,224	**5,310**
	$t=3$	5	151,773	4,368	**2,268**
	$t=4$	4	4	4	4
	$t=5$	4	4	4	4
1-out-of-12	$t=1$	11	11	11	11
	$t=4$	5	†	35,149	**6,274**
1-out-of-16	$t=1$	15	15	15	15
	$t=5$	5	5	5	5
2-out-of-16	$t=4$	8	8	8	8
	$t=5$	8	9	9	9
3-out-of-16	$t=4$	10	10	10	10

† Computation was terminated when it exceeded one hour.

TABLE IV
TOTAL NUMBER OF NONTERMINAL NODES IN ZDDS.

Benchmark functions	Compound degrees	k	No. of ZDDs	No. of nodes
1-out-of-10	$t=1$	10	9	59
	$t=2$	10	5,310	11,564
	$t=3$	10	2,268	4,765
	$t=4$	10	4	36
	$t=5$	10	4	38
1-out-of-12	$t=1$	12	11	84
	$t=4$	12	6,274	13,451
1-out-of-16	$t=1$	16	15	144
	$t=5$	16	5	67
2-out-of-16	$t=4$	120	8	939
	$t=5$	120	8	1,069
3-out-of-16	$t=4$	560	10	5,595

1) Existing [4]: the existing method [4].
2) Comm. law: the method to which only the improvement proposed in Section III-C2 is applied.
3) ZDD: the method to which both of the improvements proposed in Section III-C1 and Section III-C2 are applied.

Table III shows the number of times that the procedure *branch()* is invoked in each method for some benchmark index generation functions shown in [9]. The bold values in Table III show where the method "ZDD" significantly outperforms the other two methods.

As shown in Table III, the search space size of the methods using the commutative law of EXOR and ZDDs is a few orders of magnitude smaller than the search space of the existing one. In particular, for "1-out-of-12" with $t=4$, the Existing method could not obtain the optimum solution in a reasonable computation time because the search space was too large. However, the proposed improvement methods can obtain the optimum solution in a short computation time by avoiding redundant solution searching. From these results, we can see that the proposed improvement methods have a significant effect on reduction of the redundant solution searching.

TABLE V
COMPUTATION TIME OF METHODS (IN SECONDS).

Benchmark functions	Compound degrees	Existing [4]	Comm. law	ZDD
1-out-of-10	$t=1$	*<0.01	*<0.01	*<0.01
	$t=2$	216.62	8.56	**0.84**
	$t=3$	127.23	**0.78**	1.12
	$t=4$	0.02	*<0.01	*<0.01
	$t=5$	0.14	*<0.01	*<0.01
1-out-of-12	$t=1$	*<0.01	*<0.01	*<0.01
	$t=4$	†	30.38	**18.34**
1-out-of-16	$t=1$	*<0.01	*<0.01	*<0.01
	$t=5$	3.68	**0.04**	0.18
2-out-of-16	$t=4$	2.76	**0.14**	5.24
	$t=5$	36.32	**0.45**	14.13
3-out-of-16	$t=4$	16.17	**0.86**	207.65

* Time is less than 0.01 sec..
† Computation was terminated when it exceeded one hour.

B. On Number of Nonterminal Nodes

Table IV shows the total number of nonterminal nodes in ZDDs needed to represent all partitions of an index set that appeared during a search for solutions. These results correspond to space (memory size) complexity of the proposed method. Note that this number does not include unused nodes after ZDD operations (that is, Table IV shows the number of nodes after *garbage collection* is applied).

Since a ZDD is constructed every time *branch()* is invoked, the number of ZDDs is equal to the search space shown in Table III. As shown in Theorem 2, the number of nonterminal nodes in each ZDD is equal to the number of indices k, and thus, the number of nonterminal nodes is

$$(\text{the number of ZDDs}) \times k, \tag{1}$$

unless nonterminal nodes are shared among ZDDs. However, nonterminal nodes *are typically shared* among ZDDs. Thus, the actual number of nonterminal nodes is much smaller than (1). This means that not only one partition but also a number of partitions are represented efficiently using ZDDs.

C. On Computation Time

Although the proposed improvements reduce search space significantly, as shown in Table III, computational overhead can negate such improvements. To show that the overhead is small enough and reduction of search space leads to shortening of computation time, we compare computation time of the three methods shown in Section IV-A. Table V shows computation time, in seconds, of the three methods for the same benchmark functions.

Overheads of ZDD operations are larger than the improved overheads achieved using the commutative law. Thus, for "1-out-of-10" with $t=3$, the method using ZDDs is slower than the method using only the commutative law. This is because, for this function, search space is not much reduced using ZDDs. When search space is not reduced at all, such as "2-out-of-16" with $t=4$ and "3-out-of-16" with $t=4$, the method using ZDDs is slower than the Existing one. For "3-out-of-16" with $t=4$, the method using ZDDs is much slower than the

Existing one because, for this function, many unused nodes occur after ZDD operations, such as *Change* and *Union*, and *garbage collection* is applied repeatedly.

However, overheads of ZDD operations are not so large that the benefit of reduction in search space is canceled out, due to various techniques for ZDD operations, such as *unique table* and *computed table*. Thus, when search space size is reduced sufficiently, the method using ZDDs is the fastest. In particular, for "1-out-of-12" with $t = 4$, the method using ZDDs quickly finds the optimum solution. The Existing method was not able to find the optimum solution for this function because the search space was too large. Therefore, the proposed method using ZDDs is promising for the problem of finding optimum solutions of large index generation functions.

V. Conclusion and Comments

This paper proposes a method using ZDDs to exactly minimize the number of compound variables for linear decomposition of index generation functions. Since ZDDs represent partial solutions of the problem compactly and uniquely, the proposed method can avoid redundant solution search efficiently by reusing partial solutions. In addition, by taking advantage of techniques for ZDD operations, the proposed method can reduce the search space size of solutions significantly with small overhead, and find optimum solutions quickly. Experimental results show that the proposed method can find the optimum solution for a benchmark function that the existing method could not find.

The proposed method uses ZDDs to prune a redundant search space based on a branch and bound method. However, we think it is possible to devise a dynamic programming based method, since the number of constructed ZDDs during solution search is not large as shown in Table IV. We will study such a method as our future work.

Acknowledgments

This research is partly supported by the JSPS KAKENHI Grant (C), No.16*K*00079, and Hiroshima City University Grant for Academic Research (General Studies), 2018. The reviewers' comments were helpful in improving the paper.

References

[1] R. J. Lechner, "Harmonic analysis of switching functions," in A. Mukhopadhyay (ed.), *Recent Developments in Switching Theory*, Academic Press, New York, Chapter V, pp. 121–228, 1971.

[2] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *Proc. 30th Design Automation Conference*, pp. 272–277, 1993.

[3] S. Nagayama, T. Sasao, and J. T. Butler, "An efficient heuristic for linear decomposition of index generation functions," *46th International Symposium on Multiple-Valued Logic*, pp. 96–101, May, 2016.

[4] S. Nagayama, T. Sasao, and J. T. Butler, "An exact optimization algorithm for linear decomposition of index generation functions," *47th International Symposium on Multiple-Valued Logic*, pp. 161–166, May, 2017.

[5] E. I. Nechiporuk, "On the synthesis of networks using linear transformations of variables," *Dokl, AN SSSR*, Vol. 123, No. 4, pp. 610–612, Dec., 1958 (in Russian).

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

[7] T. Sasao, "Index generation functions: recent developments (invited paper)," *41st International Symposium on Multiple-Valued Logic*, pp. 1–9, May 2011.

[8] T. Sasao, "Linear transformations for variable reduction," *Reed-Muller Workshop 2011*, May 2011.

[9] T. Sasao, "Linear decomposition of index generation functions," *17th Asia and South Pacific Design Automation Conference*, pp. 781–788, Jan. 2012.

[10] T. Sasao, Y. Urano, and Y. Iguchi, "A lower bound on the number of variables to represent incompletely specified index generation functions," *44th International Symposium on Multiple-Valued Logic*, pp. 7–12, May 2014.

[11] T. Sasao, Y. Urano, and Y. Iguchi, "A method to find linear decompositions for incompletely specified index generation functions using difference matrix," *IEICE Transactions on Fundamentals*, Vol. E97-A, No. 12, pp. 2427–2433, Dec. 2014.

[12] T. Sasao, "A reduction method for the number of variables to represent index generation functions: s-min method," *45th International Symposium on Multiple-Valued Logic*, pp. 164–169, May 2015.

[13] T. Sasao, I. Fumishi, and Y. Iguchi, "A method to minimize variables for incompletely specified index generation functions using a SAT solver," *International Workshop on Logic and Synthesis*, pp. 161–167, June 2015.

[14] T. Sasao, I. Fumishi, and Y. Iguchi, "On an exact minimization of variables for incompletely specified index generation functions using SAT," *Note on Multiple-Valued Logic in Japan*, Vol.38, No.3, pp. 1–8, Sept. 2015 (in Japanese).