Synthesis Algorithm of Parallel Index Generation Units

Yusuke Matsunaga
Department of Advanced Information Technology,
Graduate School of Information Science and Electrical Engineering,
Kyushu University
Email: matsunaga@ait.kyushu-u.ac.jp

***Abstract*—The index generation function is a multi-valued logic function which checks if the given input vector is a registered or not, and returns its index value if the vector is registered. If the latency of the operation is critical, dedicated hardware is used for implementing the index generation functions. This paper proposes a method implementing the index generation functions using parallel index generation units. A novel and efficient algorithm called 'conflict free partitioning' is proposed to synthesis parallel index generation units. Experimental results show the proposed method outperforms other existing methods.**

***Keywords*—index generation function, logic synthesis**

I. Introduction

The index generation function is a multi-valued logic function which checks if the given input vector is a registered or not, and returns its index value if the vector is registered. If the latency of the operation is critical, dedicated hardware is used for implementing the index generation functions. Examples of such a case are address tables in the internet, terminal access controllers for local area networks, databases, memory patch circuits, electronic dictionaries, password lists, etc.[6], [9].

Sasao proposed index generation units (IGUs) to implement the index generation functions. However, a single IGU is not efficient in the sense of total memory size. So, couple of extensions are proposed. One of the methods is 'parallel sieve method'[4], which uses multiple IGUs in parallel to reduce the total memory size. Though it is efficient than single IGU, no effective algorithms that partition vectors into each IGU are known. This paper proposes a novel algorithm to partition vectors such that in each IGU, no vectors do not conflict with each other. This algorithm employs maximum bipartite matching, so it is scalable against the data size. The experimental results show that the proposed method can synthesize the parallel index generation units more efficiently than existing methods.

The rest of the paper is organized as follows: In Section II, preliminaries and the existing methods are described. Section III proposes the new method — conflict free partitioning —, SectionIV describes other related works and Section V shows the experimental results comparing with other methods. Finally, Section VI concludes the paper.

978-3-9815370-2-4/DATE14/©2014 EDAA

II. Preliminaries

A. The Index generation functions

***Definition** 1:* Consider a set of k binary vectors of n-bits R, i.e. $R \subseteq \{0,1\}^n, |R| = k$. Suppose that each vector $v_i \in R, (0 < i \leq k)$ corresponds to an integer value i. The index generation function F is a multi-valued logic function with n binary inputs and an output of range 0 to k, such that if the input vector is equal to v_i then it returns i, otherwise it returns 0. A set of vectors R is called 'the registered vectors' of F. The number of the registered vectors (k) is called 'the weight' of the index generation function. □

***Example** 1:* Table I shows an example of registered vectors and those indices. The index generation function F

TABLE I. An example of registered vector table

Vector				Index
x_1	x_2	x_3	x_4	
0	1	1	0	1
0	0	1	0	2
1	1	0	1	3
0	1	1	1	4

corresponding to the registered vector table of Table I returns 1 if $(0,1,1,0)$ is given as an input, and returns 0 if $(0,1,0,1)$ is given since $(0,1,0,1)$ is not a registered vector. □

B. Previous methods to implement the index generation functions

Since the index generation functions are just logic functions with n inputs and $\lceil \log_2(k+1) \rceil$ outputs. Recall that the special valueof '0' should be included in the output values. We might use ordinal logic synthesis methods. However, this is not realistic for a couple of reasons. First, normal logic synthesis method is not good for *random* functions. Second, from the application point of view, the registered vectors may change, and fixed gate circuits do not handle that situation. Reconfigurable circuits such as FPGAs could handle, though they have overheads of resynthesizing the circuits.

A naive way to implement the index generation functions is to use memory with n address lines. Of course, this also is not a good way. In many cases, the number of registered vectors ($= k$) is far less than 2^n, so there would be a lot of '0' entries in the memory.

Sasao proposed a couple of smart methods to implement the index generation functions[5], [12], [9], [10]. Those methods uses two blocks of memory. The first one (called 'main memory' in the paper) is used for *predicting* the index. The second one (called 'AUX memory' in the paper) is used for *checking* the predicted index. Figure 1 shows the index generation unit (IGU) described in the paper [9]. F is called

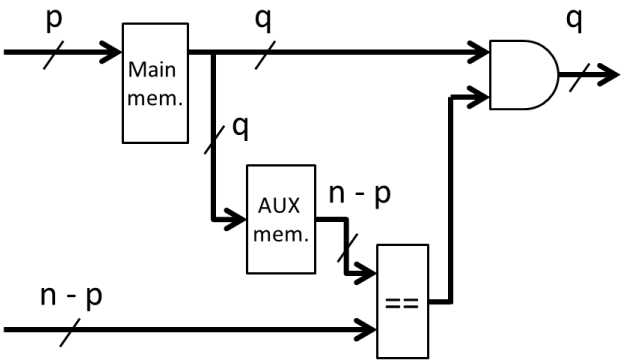

Fig. 1. The Index Generation Unit

'input hash function' having n inputs and p outputs. The purpose of input hash function is to uniformly distribute registered vectors into 2^p Boolean space. However, composing F from complex logic circuits has negative impact both on area and delay. Generally, F is composed using wires or XOR gates. In this paper, we assume input hash function is composed in similar way. The main memory has p inputs and $\lceil \log_2(k+1) \rceil$ outputs. As described above, the main memory *predicts* the index corresponding to the input. If we need an exact prediction, p would be n, that makes no sense. So some relaxation is needed. In Sasao's methods, the main memory produces a correct result if an input is a registered vector, but it produces an incorrect result if an input is not a registered vector. The AUX memory has $\lceil \log_2(k+1) \rceil$ inputs and $n-p$ outputs. The purpose of the AUX memory is to check the predicted index is correct. We only need $n-p$ bits instead of n bits, because from the predicted value of the index, we know the p bits part of the vector. If the input matches with the result of the AUX memory, the predicted index is correct, so that '1' is fed into the AND gate. On the other hand, if the input does not match with the result of the AUX memory, the predicted index is not correct, so that '0' is fed into the AND gate, and the final output becomes '0'. The number p is determined such that with p inputs, we can distinguish all the registered vectors. Solving the minimum set covering problem, we can derive the smallest p. More details are described in the paper [9]. The bit size of the main memory is $2^p \times q$, where q stands for $\lceil \log_2(k+1) \rceil$. And the bit size of the AUX memory is $2^q \times (n-p)$. The total size is shown the Equation 1.

$$2^p \times q + 2^q \times (n-p) \tag{1}$$

In [7], the following conjecture is presented.

***Conjecture** 1:* [7] Consider a set of uniformly distributed index generation functions with weight k. In most cases, an index generation function can be represented by an IGU with the main memory having at most $p = 2\lceil \log_2(k+1) \rceil - 1$ inputs.

That means, the memory size of single IGU increases in $O(k^2)$, where k is the number of registered vectors. To prevent from quadratic explosion, Nakahara et al. proposes 'the parallel sieve method'[4], which is shown in Figure2.

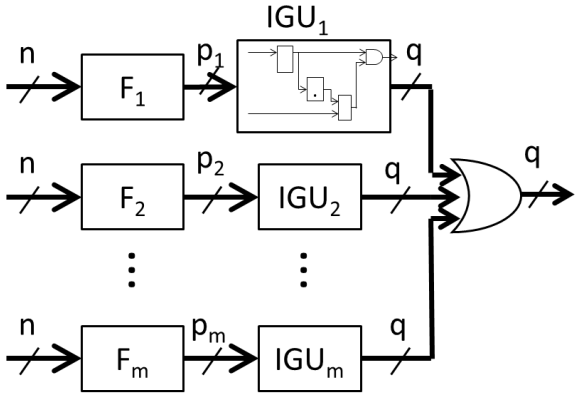

Fig. 2. Parallel sieve method

In this method, each IGU realizes a part of the registered vectors. The output of each IGU is simply bitwised OR'ed, since there is at most one IGU which has non-zero output value. Partitioning the registered vectors reduces the total memory size drastically. Originally, the size of each IGU (p_i in Fig.2) is different[4], which is complex and difficult to implement. Later on, another version using IGUs with the same size is proposed[13]. Eq.2 shows he total memory size of parallel IGU (PIGU), where m stands for the number of IGUs.

$$2^p \times (n-p+q) \times m \tag{2}$$

Both of the papers do not present any concrete method to partition the registered vectors into subgroups. Only probabilistic analysis results are shown. In [8], the following theorem is presented.

***Theorem** 1:* [8] Consider an index generation function with weight k. Then, more thatn 99.98% of the registered vectors can be realized by 4 IGUs with same size, where the number of input variables to the main memory for each IGU is $p = \lceil \log_2((k+1)/3) \rceil + 1$.

Notice that this is true is we partition the registered vectors randomly without any consideration, however, with a smart algorithm, we can reduce the total memory size further, which is shown in the next section.

III. The Proposed Method: Conflict Free Partitioning

In this section, a novel partitioning algorithm is presented. The problem formulation is as follows.

Conflict free partitioning

Inputs:

- The registered vectors R to be partitioned.
- Input hash functions $F_1, F_2, \ldots, F_m$.

Outputs:

- Partitioned vectors $R_1, R_2, \ldots, R_m$ such that $\forall k \in \{1, 2, \ldots m\}, \forall v_i, v_j \in R_k (v_i \neq v_j), F_k(v_i) \neq F_k(v_j)$.

To solve the problem, at first we build a bipartite graph G in the following way. In the graph, one group of vertices (say V_1) corresponds to the registered vectors, and the other group of vertices (say V_2) corresponds to the values of inputs hash functions. The edge of the graph corresponds to the relation between a vector and the related value of a input hash function. For example, let d_i be one of the registered vectors, and F_j be one of the input hash functions, and $s = F_j(d_i)$. There is an edge between a vertex corresponding to d_i and a vertex corresponding to s. Notice that the values of different input hash functions are to be distinguished, i.e. there is a vertex for each input hash function even if the value is the same.

The entire algorithm is in the following.

1) Construct the bipartite graph G from R and $F_1, F_2, \ldots, F_m$.
2) Find the maximum matching of G.
3) If the size of the matching is equal to the size of the vector set, then we have a partition. Otherwise, these vectors cannot be partitioned.
4) Assign each vector d into subgroup according to the matching edge.

Each vertex of V_1 has m edges. If there is a matching whose size is equal to $|V_1|$, Each vertex of V_1 has exactly one edge which is contained in the match. This edge show the assignment to which group the vector should belong. The maximum matching of bipartite graph is known to be solved in polynomial time, so this algorithm is very efficient. Notice that each vertex in V_1 has exactly m edges, where m is the number of IGUs. Usually, m is very small, that means this bipartite graph is very sparse.

***Example** 2:* Let $d_1, d_2, \ldots, d_7$ be the registered vectors, and F_1, F_2 be the input hash functions whose truth tables are shown in Table II.

TABLE II. THE TRUTH TABLE OF F_1 AND F_2

	F_1	F_2
d_1	1	1
d_2	2	2
d_3	3	3
d_4	4	3
d_5	2	4
d_6	2	1
d_7	4	1

Neither F_1 nor F_2 cannot distinguish all the registered vectors alone. Furthermore, there are triple conflicts in F_1 for d_2, d_5 and d_6 and in F_2 for d_1, d_6, and d_7. But, if we partition the vectors into two groups, $\{d_3, d_6, d_7\}$ and $\{d_1, d_2, d_4, d_5\}$(Tab.III), F_1 can distinguish the former group, and F_2 can distinguish the later group.

TABLE III. PARTITION RESULTS

	F_1
d_3	3
d_6	2
d_7	4

	F_2
d_1	1
d_2	2
d_4	3
d_5	4

To solve this problem systematically, first we build a bipartite graph in the way described above. The graph is shown in Fig. 3. In the graph, we have the maximum matching with size 7 (thick line), which is equal to the size of the vectors. And the matching tells the partitioning. For example, the edge in the matching connected to d_1 is also connected to $F_2 : 1$, which means d_1 goes into the group related to F_2. The matching guarantees that only one edge is selected for each vertex in V_2, that means there are no conflicts of output value of the input hash functions. □

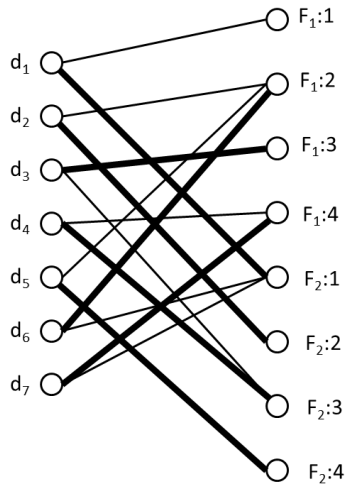

Fig. 3. Bipartite graph of the example

IV. RELATED WORK

A. Minimum perfect hash function

In the area of software algorithms, there is a similar concept called, the minimum perfect hash function (MPHF) [3], [2], [1]. The idea of perfect hashing can be applied to hardware implementation. Fig.4 shows the architecture using MPHF. In the figure, $F_i(i = 1, 2, \ldots, m)$ is called 'input hash function' and to be realized by simple wires (i.e. choosing p inputs out of n). $G_i(i = 1, 2, \ldots, m)$ is called 'mapping function' and to be realized by look-up table memory. Notice that the figure shows only a prediction part of IGU. Realizing the complete index generator requires AUX memory and comparator in addition.

MPHF construction problem is somehow resembles to conflict free partitioning problem.

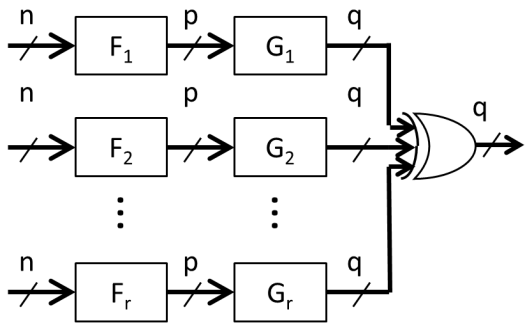

Fig. 4. The architecture of IGU using minimum perfect hash function

Minimum perfect hash function

Inputs:

- The registered vectors R to be partitioned.
- Input hash functions $F_1, F_2, \ldots, F_m$.

Outputs:

- Truth tables of the mapping functions G_i which produces conflict free (perfect) hash function.

To solve this problem, a hyper-graph called 'random graph' is constructed. A vertex of the graph corresponds to the value of each input hash function F_j with a registered vector d_i. A hyper-edge among vertices corresponds to a registered vector d_i. If the random graph is 'simple' and 'acyclic' in the sense of graph theory, then we can construct MPHF with the given vectors and input hash functions. For details, please refer[3], [2], [1].

Eq.3 shows the total memory size of IGU using MPHF.

$$2^p \times q \times m + 2^q \times n \tag{3}$$

Where n is the input bit width, p is the input size of the main memory, $q = \lceil \log_2(k+1) \rceil$, and m is the multiplexity. Notice that this equation is a little bit different from Eq.2. The bit width of the AUX memory is n, not $(n-p)$, which leads the increase of memory size if p becomes large, as we see more detail in the experimental results.

B. Row-shift decomposition

This method also generates the index value from multiple look-up table memory. Fig.5 show the architecture using row-shift decomposition[11]. This is also not an entire IGU, but only a prediction part. After the output of this circuit, AUX memory and comparator are needed. Like MPHF, F_i is an input hash function, which is realized by wires or simple logic gates. G and H are realized by look-up table memory. Unlike MPHF, the output of one memory (H) is fed into the input of another memory (G) through ADDER*. In [11], the method deriving row-shift decomposition is described. Realizing row-shift decomposition circuits, there are many choices for bit

*Actually, it is not necessary to use ADDER, but XOR gates are enough.

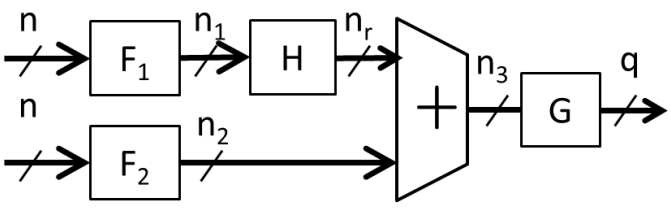

Fig. 5. The architecture of IGU using row-shift decomposition

width of each wire. For simplicity, we assume $n_3 = n_r = q$, and $n_1 = n_2 = p$. Eq.4 show the total memory size of IGU using row-shift decomposition.

$$2^p \times q + 2^q \times (q+n) \tag{4}$$

In [11], the size of AUX memory is ignored in the discussion of total memory size, which is not fair to compare other types of IGUs like parallel IGUs. So, in this paper, the size of AUX is included. The output bit width of AUX memory is also n, not $(n-p)$, since we cannot identify the p bits part of the input vector.

V. Experimental results

To evaluate the total memory size of each method (conflict free partitioning, minimum perfect hash function, and row shift decomposition). Experiments using randomly generated vectors have been done. Conflict free partitioning requires a set of input hash functions, and before generating input hash functions, p has to be fixed. However, there are no way to determine the optimal value of p. So, we start p with some proper value, and iterate to try conflict free partitioning with randomly generated input hash functions until it succeed or it reaches to the loop count limit. If it fails (i.e. reaches to the limit), increase p by one and redo the iteration. For minimum perfect hash function and row shift decomposition, the procedures are the same.

In this experiments, a set of registered vectors of 20 bits whose sizes are from 1000 to 10000 is prepared. There are 20 data for each same size and the average size is reported. The following methods are compared.

- Normal AUX memory size ($= 2^q \times n$) (AUX)
- Conflict free partitioning with $m = 2$ (CFP2).
- Conflict free partitioning with $m = 3$ (CFP3).
- Conflict free partitioning with $m = 4$ (CFP4).
- Minimum perfect hash function with $m = 2$ (MPH2).
- Minimum perfect hash function with $m = 3$ (MPH3).
- Minimum perfect hash function with $m = 4$ (MPH4).
- Row shift decomposition (RSD2).
- Parallel sieve method with $m = 4$ (PS4)

Fig.6 shows the results of the experiments. X axis stands for the size of the vectors. Y axis stands for the total memory size (in bits). In this experiments, area occupied by logic gates and wires is ignored.

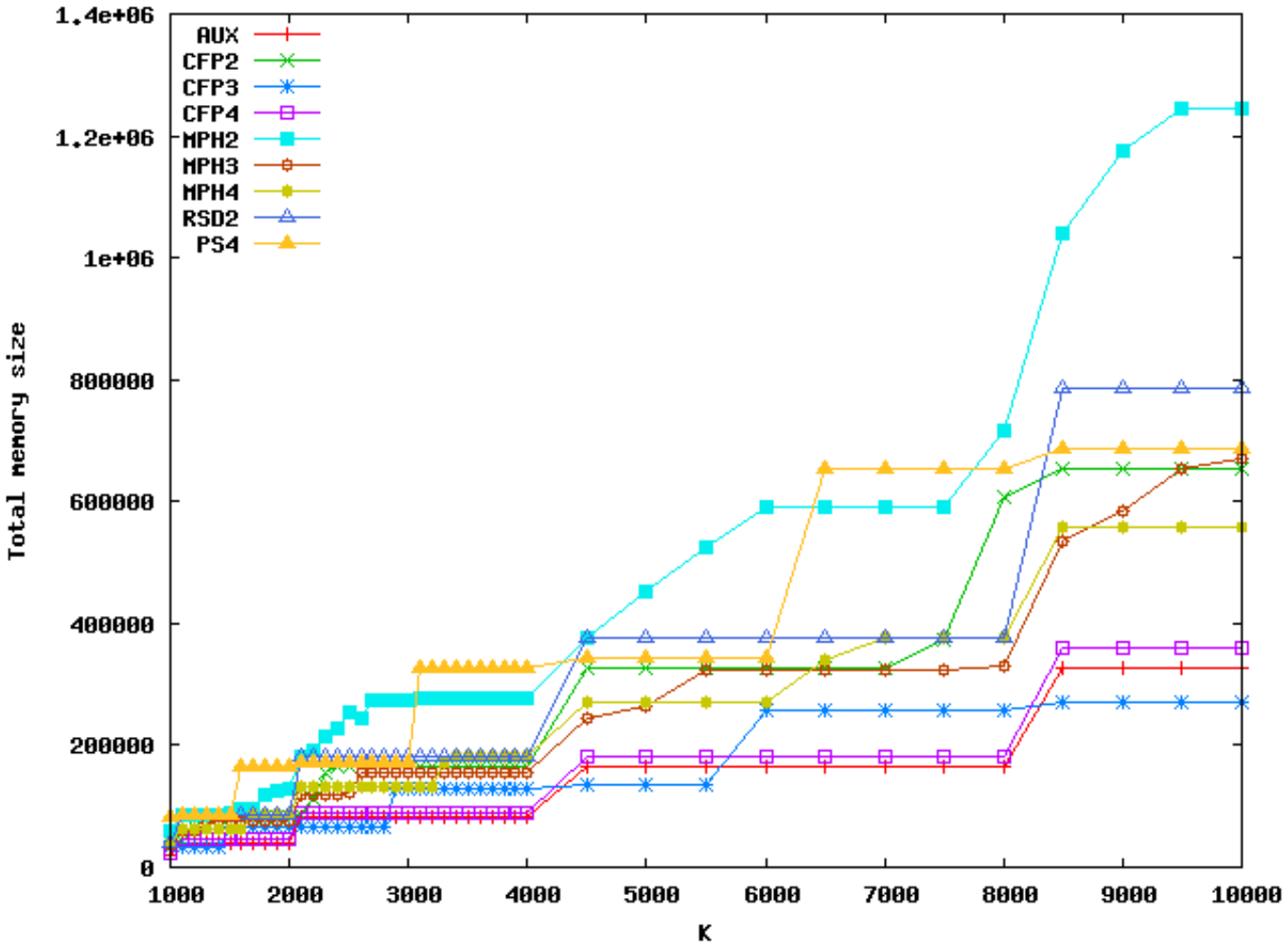

Fig. 6. Total memory size

Throughout the experiments, CFP3 and CFP4 performs well, especially when k is large. Their results are about half of others. Even when k is small, CFP3 and CFP4 are better than others. It is interesting that there is a case CFP3 is smaller than AUX. That is because the output width of the AUX memory of parallel IGU is $(n-p)$, not n, so inversion occurs according the value of p and m. Also, CFP4 is almost always very closer to AUX. Other methods, like MPH and RSD require n bits at the output of AUX memory, so they never beat when p becomes large. As described above, the current implementation of row-shift decomposition is a little bit simplified, so there are some room of improvement for the results of RSD2. However, it requires at least normal AUX memory size, so the possibility of synthesizing smaller circuits than CFP3 or CFP4 is very low. For PS4, the equation calculating the total memory size is the same with CFP4, but it requires much more memory than CFP4, which means the effect of conflict free partitioning is significant.

VI. Conclusion

A novel and effective algorithm to synthesis parallel index generation units are presented. The algorithm is based on maximum matching of bipartite graph, which is solved in polynomial time and very efficient. The experimental results show that the proposed conflict free partitioning with the multiplexity 3 or 4 performs very well. Their results are about half of other existing methods in the sense of the total memory size, and they are very closer or even smaller to the size of AUX memory only. This means that conflict free partitioning is the most robust and effective method to implement index generation functions.

Future topic includes generation of input hash functions, and extension to more complex random functions, for example, to handle with vectors having don't cares.

Acknowledgment

This research is partially supported by NEC corporation.

References

[1] Czech, Z. J., Havas, G., and Majewski, B. S. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters 43*, 5 (1992), 257 – 264.

[2] FOX, E. A., HEATH, L. S., CHEN, Q. F., AND DAOUD, A. M. Practical minimal perfect hash functions for large databases. *Commun. ACM 35*, 1 (Jan. 1992), 105–121.

[3] MAJEWSKI, B. S., WORMALD, N. C., HAVAS, G., AND CZECH, Z. J. A family of perfect hashing methods. *The Computer Journal 39*, 6 (1996), 547–554.

[4] NAKAHARA, H., SASAO, T., MATSUURA, M., AND KAWAMURA, Y. The parallel sieve method for a virus scanning engine. In *12th EUROMICRO Conference on Digital System Design, Architectures, Method and Tools (DSD-2009)* (2009), pp. 809–816.

[5] SASAO, T. A Design Method of Address Generators Using Hash Memories. In *IWLS-2006* (June 2006), pp. 102–109.

[6] SASAO, T. Design Methods for Multiple-Valued Input Address Generators. In *International Symposium on Multiple-Valued Logic (ISMVL-2006)* (May 2006).

[7] SASAO, T. On the numbers of variables to represent sparse logic functions. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design* (2008), ICCAD '08, IEEE Press, pp. 45–51.

[8] SASAO, T. On the numbers of variables to represent multi-valued incompletely specified functions. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on* (2010), pp. 420–423.

[9] SASAO, T. Index generation functions: Recent developments. In *the 41st IEEE International Symposium on Multiple-Valued Logic* (2011), pp. 1–9.

[10] SASAO, T. Linear decomposition of index generation functions. In *Proceedings of Asia and South Pacific Design Automation Conference 2012* (2012), pp. 781–788.

[11] SASAO, T. Row-shift decompositions for index generation functions. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012* (2012), pp. 1585–1590.

[12] SASAO, T., AND MATSUURA, M. An implementation of an Address Generator using Hash Memories. In *10th EUROMICRO Conference on Digital System Design, Architectures, Method and Tools (DSD-2007)* (Aug. 2007), pp. 69–76.

[13] SASAO, T., MATSUURA, M., AND NAKAHARA, H. A realization of index generation functions using modules of uniform sizes. In *19th International Workshop on Logic and Synthesis (IWLS-2010)* (June 2010), pp. 201–208.