A Realization of Index Generation Functions Using Modules of Uniform Sizes

Tsutomu Sasao, Munehiro Matsuura, and Hiroki Nakahara
Kyushu Institute of Technology, Iizuka 820-8502, Japan
April 27, 2010

***Abstract*—This paper considers a method to realize index generation functions. The parallel sieve method developed by the authors efficiently implements an index generation function. Unfortunately, it requires many Index Generation Units (IGUs) with different sizes. This paper shows a design method that requires only four IGUs with the same size. The presented architecture can be used as a low-power content addressable memory (CAM).**

I. Introduction

One of the important operations in information processing is to efficiently find desired data from a large data set. For example, consider a network router. In the case of IPv4, IP addresses are represented by 32 bits. A network router stores about 40,000 of the 2^{32} possible combinations of inputs, and checks if an input pattern matches a stored pattern [2], [3].

A content addressable memory (CAM) is a device that performs this operation directly [7]. CAMs are also used for virus scanning and spam-mail filters. An index generation function can be represented by a registered vector table such as shown in Table II. It can be implemented by a CAM [7], or FPGA [14], or a combination of memories and logic [1], [6].

In a previous paper, the authors describe the parallel sieve method [5]. It consists of basic pattern matching modules called **IGU**s (Index Generation Units). Since an IGU uses ordinary memory and some logic, the cost and the power dissipation are much lower than CAM-based implementations. We implemented a virus scanning system for 500,000 patterns by using SRAMs and an FPGA. Due to high cost and power dissipation[1], implementation of such virus scanning circuit by conventional CAMs is difficult. Unfortunately, the standard parallel sieve method uses many IGUs with different sizes. So, the update of the data is complicated. In this paper, we prove that most index generation functions can be implemented by four IGUs of uniform size. Thus, the implementation and update of the data are much simpler than in the previous method. The presented architecture can be used as a low-power content addressable memory (CAM).

II. Index Generation Function

In this part, we introduce index generation functions.

[1]In a CAM, all the cells operate in parallel: The power dissipation of a CAM cell per bit can be 150 times higher than SRAM [15]. Also, since the CAM circuit is more complicated than memory, the cost of the CAM chip can be 30 times higher than DDR SRAM [15].

TABLE 2.1
Registered vector table.

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

TABLE 2.2
Index generation function.

Input				Output		
x_1	x_2	x_3	x_4	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	**0**	**0**	**1**
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	**0**	**1**	**0**
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	**0**	**1**	**1**
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	**1**	**0**	**0**

Definition 2.1: 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 the **index** of each registered vector. An **index generation function** represents a mapping: $\{0,1\}^n \rightarrow \{0,1,2,\ldots,k\}$. It produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function.

Example 2.1: Table II shows a registered vector table with weight $k = 4$, and $n = 4$. The corresponding index generation function is shown in Table 2.1.

Here, we assume that k is much smaller than 2^n, the total number of possible input combinations. Index generation functions are used in address tables in the Internet, terminal access controller for local area networks, databases, memory patch circuits, dictionaries, password lists, etc.[9].

III. Index Generation Unit (IGU)

In this section, we show an efficient method to implement index generation functions. With this method, the number of variables to the memory can be reduced. Fig. 3.1 shows the **Index Generation Unit** (IGU). The **programmable hash circuit** has n inputs and p outputs, where $p < n$. The set of inputs to the programmable hash circuit is $X = (X_1, X_2)$, and the

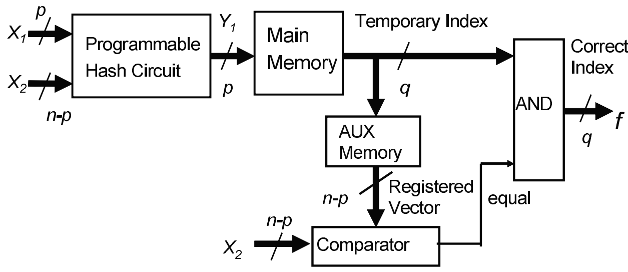

Fig. 3.1. Index generation unit (IGU).

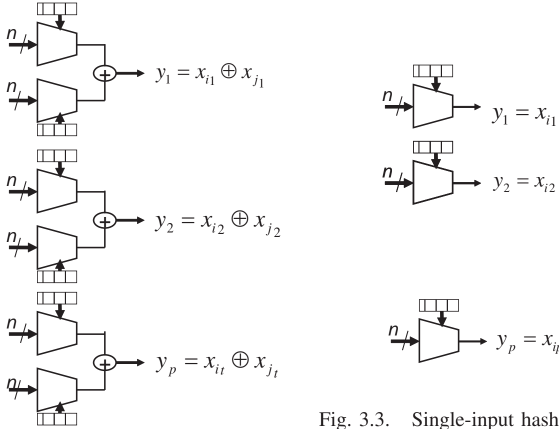

Fig. 3.3. Single-input hash circuit.

Fig. 3.2. Double-input hash circuit.

output is $Y_1 = (y_1, y_2, \ldots, y_p)$. It is used to rearrange the non-zero elements. We consider two types of programmable hash circuits. The first type is the **double-input hash circuit** shown in Fig. 3.2. It performs a **linear transformation** $y_i = x_i \oplus x_j$ or $y_i = x_i$, where $x_i \in X_1$ and $x_j \in X_2$. 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 register with $\lceil \log_2 n \rceil$ bits specifies which variable to select by the multiplexer. 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 second type of a programmable hash circuit is the **single-input hash circuit** shown in Fig. 3.3. It consists of only p multiplexers, and selects p variables from n input variables. Note that both types of ha circuits produce only specific kinds of hash functions. We have found that these functions are suitable for our application. For more discussion on this, see Section X. The **main memory** has p inputs and $q = \lceil \log_2(k+1) \rceil$ outputs. The main memory produces correct outputs for registered vectors. However, it may produce incorrect outputs for non-registered vectors, because the number of input variables is reduced to p. 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 q inputs and $(n-p)$ outputs: It stores the X_2 part of 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.

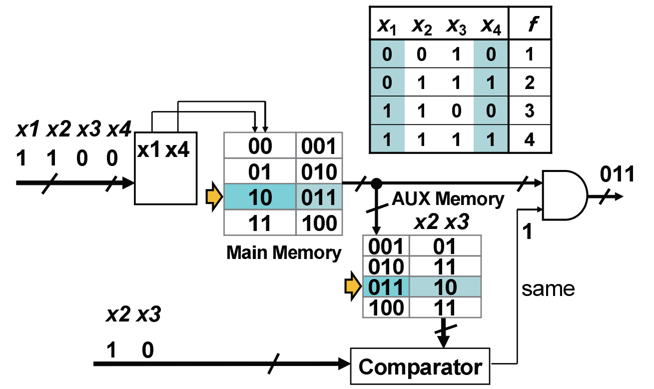

Fig. 3.4. When the input vector is registered.

Otherwise, the main memory produces a wrong output, and the input vector is non-registered. Thus, the **output AND gates** produce 0 outputs, showing that the input vector is non-registered. Note that the main memory produces the correct outputs only for the registered vectors.

Example 3.1: Consider the registered vectors in Table II. The number of variables is four, but only two variables x_1 and x_4 are necessary to distinguish these four registered vectors. Fig. 3.4 shows the IGU. In this case, the programmable hash circuit produces $Y_1 = (x_1, x_4)$ from $X = (x_1, x_2, x_3, x_4)$. The main memory stores the indices for $X_1 = Y_1 = (x_1, x_4)$, and the AUX memory stores the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector.

When the input vector is registered

Suppose that a registered vector $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$ is applied to the IGU in Fig. 3.4. First, the programmable hash circuit selects two variables, x_1 and x_4, and produces the value $X_1 = (x_1, x_4) = (1, 0)$. Second, the main memory produces the corresponding index $(0, 1, 1)$. Third, the AUX memory produces the values of $X_2 = (x_2, x_3) = (1, 0)$ corresponding registered vector $(1, 1, 0, 0)$. Fourth, the comparator confirms that the values of $X_2 = (x_2, x_3)$ of the input vector is equal to the output of the AUX memory. And, finally, the AND gate produces the index for the input vector.

When the input vector is not registered

Suppose that a non-registered vector $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ is applied to the IGU in Fig. 3.5. Also in this case, the main memory produces the vector $(0, 1, 1)$, and the AUX memory produces the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector $(1, 1, 0, 0)$. However, in this case, the comparator shows that $X_2 = (x_2, x_3) = (0, 1)$ is different from the output $X_2 = (x_2, x_3)$ of the AUX memory. Thus, the AND gate produces zero output, which indicates that the input vector is not registered. ■

Unfortunately, not all index generation functions have the nice properties of Example 3.1. So, we decompose the given function into two:

1) A function that is implemented by an IGU.
2) The remaining part.

Given an index generation function $f(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$, we decom-

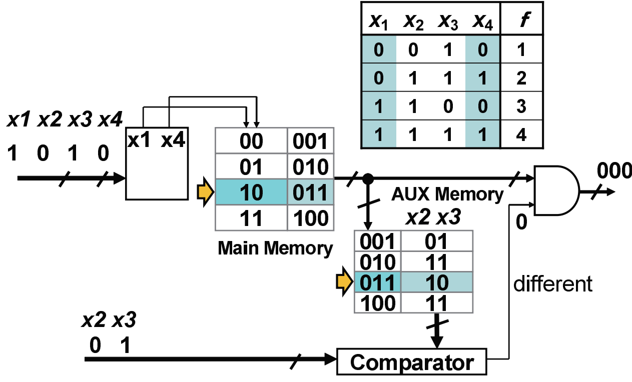

Fig. 3.5. When the input vector is not registered.

pose it into two disjoint sub-functions:

$$f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee f_2(X_1, X_2),$$

where each column of the decomposition chart [8] for $\hat{f}_1(Y_1, X_2)$ has at most one non-zero element. In this case, $\hat{f}_1(Y_1, X_2)$ can be implemented by an IGU, where the inputs to the main memory is $Y_1 = (y_1, y_2, \ldots, y_p)$. Since $f_2(X_1, X_2)$ has fewer non-zero elements than the original function, it is simpler to implement.

Theorem 3.1: Consider the IGU in Fig. 3.1. Assume that $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$ for $j \in \{p+1, p+2, \ldots, n\}$, or $y_i = x_i$, are applied to the input to the main memory. If the main memory of an IGU implements the function $\tilde{g}(Y_1)$, where $\tilde{g}(Y_1)$ produces the non-zero value if the column Y_1 of the decomposition chart for $\hat{f}_1(Y_1, X_2)$ has a non-zero value, and $\tilde{g}(Y_1) = 0$ otherwise, then only the values for X_2 must be stored in the AUX memory.

(Proof) Consider the decomposition chart of the function $\hat{f}_1(Y_1, X_2)$. By the assumption of the construction, each column of the decomposition chart has at most one non-zero element. When a registered vector is applied to the IGU, the main memory produces a non-zero output. In this case, the X_2 part of the input vector is equal to the output of the AUX memory, indicating that the vector is registered. Thus, the IGU produces the correct non-zero output.

Assume that the input vector is not registered, but the output of the AUX memory is equal to the X_2 part of the input vector. We have two cases:

1) The main memory produces the zero-output.
   In this case, even if the X_2 part of the input vector is equal to the output of the AUX memory, the output of the main memory is zero. Thus, the IGU produces the correct output.
2) The main memory produces a non-zero output.
   Due to the construction of the IGU, the input vector is registered. However, this contradicts the assumption. So, such a case never happens.

□

IV. Number of Vectors Realized by IGU

In this section, we derive the expected number of registered vectors implemented by an IGU.

TABLE 4.1
Decomposition chart for $f(X_1, X_2)$.

			0	0	0	0	1	1	1	1	x_3
			0	0	1	1	0	0	1	1	x_2
			0	1	0	1	0	1	0	1	x_1
0	0	0	0	0	0	0	0	0	0	0	
0	0	1	0	0	0	0	1	0	0	0	
0	1	0	0	0	0	0	0	2	0	0	
0	1	1	0	3	0	0	0	0	0	0	
1	0	0	0	4	0	0	0	0	5	0	
1	0	1	0	0	0	0	6	0	0	0	
1	1	0	0	0	0	0	0	0	0	0	
1	1	1	7	0	0	0	0	0	0	0	
x_6	x_5	x_4									

Lemma 4.1: When $0 < \alpha << 1$, $1-\alpha$ can be approximated by $e^{-\alpha}$.

Lemma 4.2: Let $f(X)$ be a uniformly distributed index function of n variables with weight k, where $k << 2^n$. Consider a decomposition chart, and let p be the number of bound variables. Then, the probability that a column of the decomposition chart has all-zero elements is approximately $e^{-\xi}$, where $\xi = \frac{k}{2^p}$.

(Proof) The probability that a function takes a non-zero value is $\alpha = \frac{k}{2^n}$. The probability that a function takes a zero value is $\beta = 1 - \alpha$. Since the decomposition chart has 2^{n-p} rows, the probability that a column of the chart has all zero elements is

$$\beta^{2^{n-p}} = (1-\alpha)^{2^{n-p}}$$

Since $0 < \alpha << 1$, by Lemma 4.1, $1 - \alpha$ is approximated by $e^{-\alpha}$, we have

$$\beta^{2^{n-p}} \simeq e^{-\alpha \cdot 2^{n-p}} = e^{-\frac{k}{2^p}} = e^{-\xi}$$

□

Theorem 4.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the expected number of registered vectors of f that can be realized by the IGU is $2^p(1-e^{-\xi})$, where $\xi = \frac{k}{2^p}$.

(Proof) Let (X_1, X_2) be a partition of the input variables X, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the decomposition chart for $f(X_1, X_2)$, where X_1 labels the column variables and X_2 labels the row variables. If a column has at least one non-zero element, then the IGU can realize an element of the column. From Lemma 4.2, the probability that each column has at least one non-zero element is $1 - e^{-\xi}$, where $\xi = \frac{k}{2^p}$. Since there are 2^p columns, the expected number of registered vectors realized by the IGU is $2^p(1 - e^{-\xi})$. □

Example 4.1: Table 4.1 shows the decomposition chart for a 6-variable index generation function with weight $k = 7$. Note that $X_1 = (x_1, x_2, x_3)$ denotes the bound variables, and $X_2 = (x_4, x_5, x_6)$ denotes the free variables. In this case, three columns $(x_1, x_2, x_3) = (0,1,0)$, $(0,1,1)$, and $(1,1,1)$ have all zero elements. In the other words, the fraction of columns that have all zero elements is $\frac{3}{8} = 0.375$. In Lemma 4.2, we have $n = 6$, $p = 3$, and $\xi = \frac{k}{2^p} = 0.875$. It shows that the probability that a column has all zero element is $e^{-\xi} = 0.4169$.

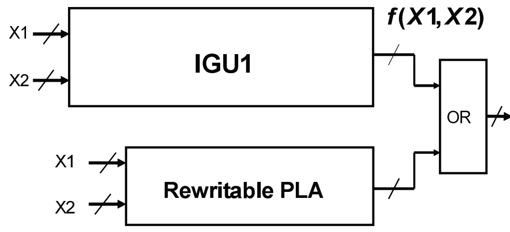

Fig. 5.1. Index generator implemented by hybrid method.

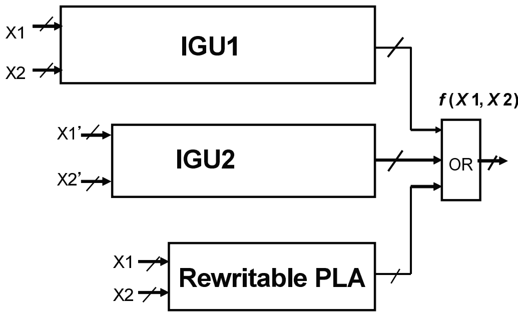

Fig. 5.2. Index generator implemented by super hybrid method.

In Theorem 4.1, the expected number of vectors realized by the IGU is

$$2^p(1-e^{-\xi}) = 8 \times 0.583 = 4.665.$$

In Table 4.1, five vectors for 7, 3, 1, 2, 5 can be realized by an IGU. The vectors for 4 and 6 should be realized by other parts of the circuit. ∎

Corollary 4.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the fraction of registered vectors of f that can be realized by the IGU is

$$\delta = \frac{1-e^{-\xi}}{\xi},$$

where $\xi = \frac{k}{2^p}$.

For example, when $\xi = \frac{1}{4}$, we have $\delta \simeq 0.8848$, when $\xi = \frac{1}{2}$, we have $\delta \simeq 0.7869$, and when $\xi = 1$, we have $\delta \simeq 0.63212$.

In [12], it is shown that when $p = 2q-1$, where $q = \lceil \log_2(k+1) \rceil$, the IGU implements most of the registered vectors. However, when k is large, the IGU requires a huge main memory, since the main memory has $2q-1$ inputs and $q = \lceil \log_2(k+1) \rceil$ outputs. For example, when $k = 500,0000$, the size of the main memory is $q2^p = 19 \times 2^{37} = 2.375$ terabits. Thus, we need a more efficient method.

V. Parallel Sieve Method

The **hybrid method** shown in Fig. 5.1 uses one IGU and a rewritable PLA [10]. In this method, the main memory has $p = q+2$ inputs, and realizes 88% of the registered vectors, where $q = \lceil \log_2(k+1) \rceil$. The rest of the registered vectors are implemented by the rewritable PLA. The **super hybrid method** shown in Fig. 5.2 uses two IGUs and a rewritable PLA [11]. In this method, the main memory in IGU1 has $p = q+1$ inputs, and realizes 79% of the registered vectors. The main memory in IGU2 has $p = q$ inputs, and realizes 16.6% of registered vectors. The rest of the registered vectors are implemented by the rewritable PLA.

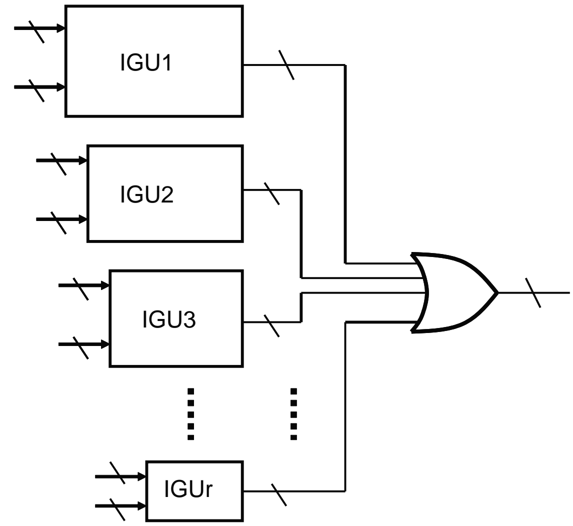

Fig. 5.3. Index generator implemented by parallel sieve method.

By increasing the number of IGU's, we have the parallel sieve method shown in Fig. 5.3. This method is especially useful when the number of the registered vectors is very large [5].

Definition 5.1: The **parallel sieve method** is an implementation of an index generation function using multiple IGUs as shown in Fig. 5.3. IGU_{i+1} is used to realize a part of the registered vectors not realized by IGU_1, IGU_2, ..., or IGU_i. The OR gate in the output combines the indices to form a single output. In the **standard parallel sieve method**, the number of inputs to the main memory is chosen as

$$p_i = \lceil \log_2(k_i+1) \rceil,$$

where k_i denotes the number of registered vectors to be realized by IGU_j, $(j \geq i)$.

Example 5.1: (Standard Parallel Sieve Method)
By using the standard parallel sieve method, realize an index generation function with $n = 40$ and $k_1 = 49151$.

1) In IGU_1, the number of inputs for the main memory is $p_1 = q_1 = \lceil \log_2(k_1+1) \rceil = 16$. By Theorem 4.1, the number of the vectors realized by IGU_1 is $2^{p_1}(1-e^{-\xi_1})$, where $\xi_1 = \frac{k_1}{2^{p_1}}$, that is $65536 \times 0.527626 = 34578$. The number of the remaining vectors is $k_2 = k_1 - 34578 = 14573$.
2) In IGU_2, since $q_2 = \lceil \log_2(14573+1) \rceil = 14$, we have $p_2 = q_2 = 14$. The number of the vectors realized by IGU_2 is $2^{p_2}(1-e^{-\xi_2})$, where $\xi_2 = \frac{k_2}{2^{p_2}}$, that is $16384 \times 0.5891246 = 9652$. The number of the remaining vectors is $k_3 = k_2 - 9652 = 4921$.
3) In IGU_3, since $q_3 = \lceil \log_2(4921+1) \rceil = 13$, we have $p_3 = q_3 = 13$. The number of vectors realized by IGU_3 is $2^{p_3}(1-e^{-\xi_3})$, where $\xi_3 = \frac{k_3}{2^{p_3}}$, that is

$8192 \times 0.4515768 = 3699$. The number of the remaining vectors is $k_4 = k_3 - 3699 = 1222$.

4) In IGU_4, since $q_4 = \lceil \log_2(1222+1) \rceil = 11$, we have $p_4 = q_4 = 11$. The number of vectors realized by IGU_4 is $2^{p_4}(1-e^{-\xi_4})$, where $\xi_4 = \frac{k_4}{2^{p_4}}$, that is $2048 \times 0.4493631 = 920$. The number of the remaining vectors is $k_5 = k_4 - 920 = 302$.
5) In IGU_5, since $q_5 = \lceil \log_2(302+1) \rceil = 9$, we have $p_5 = q_5 = 9$. The number of vectors realized by IGU_5 is $2^{p_5}(1-e^{-\xi_5})$, where $\xi_5 = \frac{k_5}{2^{p_5}}$, that is $512 \times 0.4455861 = 228$. The number of the remaining vectors is $k_6 = k_5 - 228 = 74$.
6) In IGU_6, since $q_6 = \lceil \log_2(74+1) \rceil = 7$, we have $p_6 = q_6 = 7$. The number of vectors realized by IGU_6 is $2^{p_6}(1-e^{-\xi_6})$, where $\xi_6 = \frac{k_6}{2^{p_6}}$, that is $128 \times 0.4390509 = 56$. The number of the remaining vectors is $k_7 = k_6 - 56 = 18$.
7) In IGU_7, since the number of the remaining vectors is only $k_7 = 18$, they can be implemented by an IGU [12], or rewritable PLA or an LUT cascade. ■

Note that, in IGU_i, the main memory has p_i inputs and p_i outputs, while the AUX memory has p_i inputs and $(n-p_i)$ outputs. Thus, the total amount of memory for IGU_i is

$$p_i 2^{p_i} + (n-p_i)2^{p_i} = n2^{p_i}.$$

Let u be the number of the IGUs. Then, the total memory for the standard parallel sieve method is

$$n \sum_{i=1}^{u} 2^{p_i}.$$

VI. Parallel Sieve Method using Uniform Sizes of IGUs

The standard parallel sieve method efficiently implements index generation functions. Unfortunately, it requires many IGUs with different sizes. This is inconvenient for the update of registered vectors. In this section, we show that most index generation functions can be realized with only four IGUs with the same size.

Theorem 6.1: Consider an index generation function with weight k. Then, more than 99.98% of the registered vectors can be realized by the architecture shown in Fig. 6.1, where the number of input variables to the main memory for each IGU is $p = \lceil \log_2((k+1)/3) \rceil + 1$.

(Proof) Let $k_1 = k$. We assume that for each IGU, the distribution of the vectors is uniform. This can be accomplished by careful design of programmable hash circuits.

1) In IGU_1: Let $\xi_1 = \frac{k_1}{2^p}$.
The number of realized vectors is $2^p(1-e^{-\xi_1})$.
The number of remaining vectors is
$k_2 = k_1 - 2^p(1-e^{-\xi_1}) = k_1 + 2^p(e^{-\xi_1} - 1)$.
2) In IGU_2: Let $\xi_2 = \frac{k_2}{2^p} = \frac{k_1}{2^p} + (e^{-\xi_1} - 1)$.
The number of realized vectors is $2^p(1-e^{-\xi_2})$.
The number of remaining vectors is

$$\begin{aligned} k_3 &= k_1 - 2^p(1-e^{-\xi_1}) - 2^p(1-e^{-\xi_2}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} - 2). \end{aligned}$$

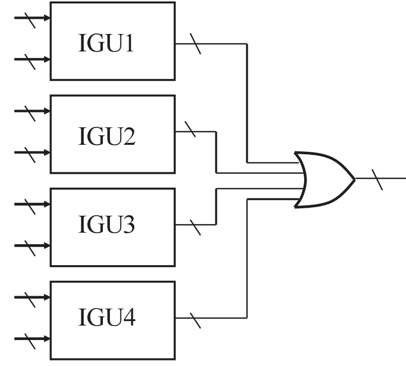

Fig. 6.1. Realization of an index generation function using 4 IGUs.

3) In IGU_3: Let $\xi_3 = \frac{k_3}{2^p} = \frac{k}{2^p} + (e^{-\xi_1} + e^{-\xi_2} - 2)$.
The number of realized vectors is $2^p(1-e^{-\xi_3})$.
The number of remaining vectors is

$$\begin{aligned} k_4 &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} - 2) - 2^p(1-e^{-\xi_3}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3). \end{aligned}$$

4) In IGU_4: Let $\xi_4 = \frac{k_1}{2^p} + (e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3)$.
The number of realized vectors is $2^p(1-e^{-\xi_4})$.
The number of remaining vectors is

$$\begin{aligned} k_5 &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3) - 2^p(1-e^{-\xi_3}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} + e^{-\xi_4} - 4). \end{aligned}$$

5) The fraction of remaining vectors is
$\frac{k_5}{2^p} = \frac{k_1}{2^p} + (e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} + e^{-\xi_4} - 4)$.
When $k_1 = 2^p + 2^{p-1}$, the fraction is about 1.42×10^{-4}, and is sufficiently small. □

Example 6.1: (Parallel Sieve Method with Uniform IGU sizes)
Consider an index generation function with $n = 40$ and $k = 49151$. Let us realize the function by the architecture shown in Fig. 6.1. Suppose that the number of inputs to the main memory in each IGU is $p = 15$. We assume that for each IGU, the distribution of the vectors is uniform. This can be accomplished by tuning the programmable hash circuit.

1) In IGU_1: Let $\xi_1 = \frac{k}{2^p} = \frac{49151}{2^{15}} = 1.5$. By Theorem 4.1, it realizes $2^p(1-e^{-\xi_1}) = 32768 \times 0.776863 \simeq 25,456$ registered vectors. The number of remaining vectors is $k_2 = 23695$.
2) In IGU_2: Let $\xi_2 = \frac{k_2}{2^p} = \frac{23695}{2^{15}} = 0.723114$. By Theorem 4.1, it realizes $2^p(1-e^{-\xi_2}) = 32768 \times 0.5147611 \simeq 16867$ registered vectors. The number of remaining vectors is $k_3 = 6828$.
3) In IGU_3: Let $\xi_3 = \frac{k_3}{2^p} = \frac{6828}{2^{15}} = 0.208374$. By Theorem 4.1, it realizes $2^p(1-e^{-\xi_3}) = 32768 \times 0.1880967 \simeq 6163$ registered vectors. The number of remaining vectors is $k_4 = 665$.
4) In IGU_4: Let $\xi_4 = \frac{k_4}{2^p} = \frac{665}{2^{15}} = 0.0202942$. By Theorem 4.1, it realizes $2^p(1-e^{-\xi_4}) = 32768 \times 0.0202942 \simeq 658$ registered vectors. The number of remaining vectors is only $k_5 = 7$. ■

TABLE 7.1
REGISTERED VECTOR TABLE FOR 6-VARIABLE FUNCTION .

Vector						Index
x_1	x_2	x_3	x_4	x_5	x_6	
1	0	0	0	0	0	1
0	1	0	0	0	0	2
0	0	1	0	0	0	3
0	0	0	1	0	0	4
0	0	0	0	1	0	5
0	0	0	0	0	1	6
0	0	0	0	0	0	7

The above example shows that almost all the vectors can be implemented by four IGUs. In the next section, we show a method to implement all the vectors on four IGUs.

VII. DESIGN OF PROGRAMMABLE HASH CIRCUITS

To realize index generation functions by the parallel sieve method, the design of the programmable hash circuits is vitally important. Consider the following:

Example 7.1: In the index table in Table 7.1, the number of 0's is much larger than that of 1's. In this case, all the variables are necessary to represent the function, since any change of each variable from $(0,0,0,0,0,0)$ will change the value of the function.

1) A single-input hash circuit is used.
   Since all the variables are essential, the main memory requires 6 variables.
2) A double-input hash circuit is used.
   Consider the transform:

$$y_1 = x_1 \oplus x_5$$
$$y_2 = x_2 \oplus x_5$$
$$y_3 = x_3 \oplus x_6$$
$$y_4 = x_4 \oplus x_6$$

   Table 7.2 shows the transformed function. In this case, all the patterns are different. This means that these four variables are sufficient to represent the function. In fact, this is a minimum solution when a double-input hash circuit is used.
3) A triple-input hash circuit is used.
   Consider the transform:

$$z_1 = x_1 \oplus x_5 \oplus x_6$$
$$z_2 = x_2 \oplus x_4 \oplus x_6$$
$$z_3 = x_3 \oplus x_4 \oplus x_5$$

   Table 7.3 shows the transformed function. In this case, all the patterns are different. This means that three variables are sufficient to represent the function. In fact, this is a minimum solution when a hash circuit with any number of inputs is used.

■

We use the following strategy to design the programmable hash circuits:

- Use both single and double input hash circuits.
- In each IGU, maximize the number of vectors implemented by the IGU.

TABLE 7.2
INDEX TABLE FOR IGU WITH DOUBLE-INPUT HASH CIRCUIT.

Vector				Index
y_1	y_2	y_3	y_4	
1	0	0	0	1
0	1	0	0	2
0	0	1	0	3
0	0	0	1	4
1	1	0	0	5
0	0	1	1	6
0	0	0	0	7

TABLE 7.3
INDEX TABLE FOR IGU WITH TRIPLE-INPUT HASH CIRCUIT.

Vector			Index
z_1	z_2	z_3	
1	0	0	1
0	1	0	2
0	0	1	3
0	1	1	4
1	0	1	5
1	1	0	6
0	0	0	7

From here, we present a method to design programmable hash circuits. To find the optimum setting of the programmable hash circuits, we use the following:

Algorithm 7.1:
1) Let f be the index generation function of n variables with weight k. Let $p = \lceil \log_2((k+1)/3) \rceil + 1$ be the number of the bound variables in the decomposition chart.
2) Let the bound set $\{X_1\}$ be the initial set of essential variables.
3) While $|X_1| \leq p$, find the non-essential variables x_i that makes the following value minimum:

$$|(\text{\# of vectors with } x_i = 0) - (\text{\# of vectors with } x_i = 1)|.$$

   $\{X_1\} \leftarrow \{X_1\} \cup \{x_i\}$.
4) Let $X_1 = (x_1, x_2, \ldots, x_p)$ be the bound variables, and let $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$ be the free variables.
5) For each pair of variables (x_i, x_j), where x_i is a bound variable, and x_j is a free variable, if the exchange of x_i with x_j increases the column multiplicity, then do it, otherwise discard it.
6) For each pair of variables (x_i, x_j), where x_i is a bound variable, and x_j is a free variable, if the replacement of x_i with $y_i = x_i \oplus x_j$ increases the column multiplicity, then do it, otherwise discard it.

VIII. UPDATE OF REGISTERED VECTORS

In some applications, the registered vectors must be updated frequently. In our architecture, only memory data must be updated, since interconnections are fixed.

An update of registered vectors can be performed by a series of two operations: Deletion of a vector, and addition of a vector [4]. A **deletion of a vector** is simple: just remove the corresponding elements from the main memory and the AUX memory. An **addition of a vector** is more complicated. If the vector can be put into a vacant column of the decomposition chart of any of the IGUs, then we can add the vector to the main memory and the AUX memory. Otherwise, we have to re-design all the IGUs, which requires longer time. The frequency of the re-design can be reduced by adding a small rewritable PLA as the fifth IGU.

TABLE 9.1
NUMBERS OF REALIZED VECTORS BY IGUS (RANDOM FUNCTIONS WITH WEIGHT 49151)

IGU	Estimated		Experimental	
	k_i	Realized Vectors	k_i	Realized Vectors
1	49151	25456	49151.00	25633.91
2	23695	16867	23517.09	16929.49
3	6828	6163	6587.60	6530.77
4	665	658	556.83	556.82
Total		49144		49150.99

TABLE 9.2
NUMBERS OF REALIZED VECTORS BY IGUS (IP ADDRESS TABLE)

IGU	Estimated		Experimental	
	k_i	Realized Vectors	k_i	Realized Vectors
1	7903	5070	7903	5267
2	2833	2395	2636	2329
3	438	426	307	307
4	12	11	0	0
Total		7902		7903

IX. EXPERIMENTAL RESULTS

We used two types of data to show the usefulness of the approach experimentally.

A. Randomly Generated Functions

We generated uniformly distributed index generation functions of $n = 32$ variables, and implemented functions by a parallel sieve method of uniform IGU sizes. Table 9.1 shows the numbers of vectors realized by four IGUs. One hundred index generation functions with $k = 49151$ registered vectors were generated. For the other combinations, the outputs are set to zeros. The column headed *Estimated* denotes ones that were obtained by Theorem 4.1. The column headed *Experimental* denotes the average of 100 randomly generated functions. In the experiment, only single-input hash circuits were used. In the experiment, more vectors could be implemented than estimated values. This is because hash functions are selected to realize more vectors in IGU1, IGU2, and IGU3. Out of 100 functions, 99 functions were implemented by 4 IGUs. If we use both single and double-input hash circuits, then all the vectors can be implemented. We did similar experiments for $k = 1536, 3971, 6143, 12287$, and 24575, and obtained similar results. In these cases, only single-input hash circuits were used. The tables are omitted due to the space limitation.

B. IP Address Table

We also used IP addresses of computers that accessed our WEB site in a certain period. The table contains 7903 unique addresses. The number of inputs is $n = 32$, and the number of outputs is 13. Estimation required four IGUs, while the experiment required only three IGUs using both single-input and double-input hash circuits. In the case of IP address table, the distribution of vectors were not uniform, and double-input hash circuits were necessary.

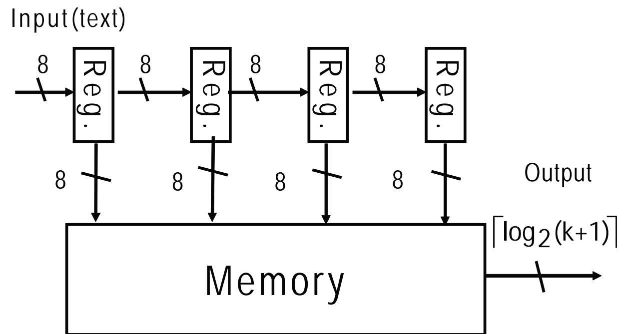

Fig. 9.1. Virus scanning circuit.

C. Virus Scanning System

We have developed a system that detects viruses [5]. A complete system using only hardware is too complex, so we used two-stage method: In the first stage, suspicious patterns are detected by hardware, and in the second stage, a complete match is performed by software only for the patterns detected in the first stage. Here, we consider the hardware part in the first stage. We check the text using a window of four characters, and the number of suspicious patterns is $k = 497,172$. Fig. 9.1 shows the circuit to detect the suspicious patterns. Eight 4-stage shift registers are used to store four characters. These registers works as a window. Note that the number of inputs to the memory is $4 \times 8 = 32$, and the number of outputs is $\lceil \log_2(k+1) \rceil = 19$. A straightforward implementation requires a memory with impractical size: $\lceil \log_2(k + 1) \rceil 2^{32} = 76$G bits. If we use parallel sieve method shown in Section VI, we need only 64M bits. In this case, single-input hash was not sufficient; a double-input hash circuit was necessary to store all the patterns into four IGUs.

X. CONCLUSION AND COMMENTS

In this paper, we presented a method to implement index generation functions. We show that most index functions can be realized with four IGU of equal sizes, where the number of inputs to the main memory of each IGU is $p = \lceil \log((k + 1)/3) \rceil + 1$. Experimental results using randomly generated functions, IP address, and virus detection system confirmed the validity of the approach.

In the programmable hash circuits, we use only single-input and double-input hash circuits. However, we can use hash circuits with more inputs. We considered the usefulness of triple-input hash circuits in [13]. A heuristic method to find a good linear transformation has been developed. In fact, the use of triple-input hash was effective when the given functions have a very large skew as shown in Example 7.1. However, triple-input hash circuits are rather expensive to implement. In the current applications, we consider that two-input hash circuits are sufficient.

ACKNOWLEDGMENTS

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

Research, and the MEXT Regional Innovation Cluster Program (Global Type, 2nd Stage). Reviewer's comments significantly improved the presentation of the paper. Discussion with Prof. Jon T. Butler was quite useful.

REFERENCES

[1] Y. H. Cho and W. H. Mangione-Smith, ”Deep packet filter with dedicated logic and read only memories,” *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (FCCM'04), pp.125-134.

[2] P. Gupta, S. Lin , and N. McKeown, “Routing lookups in hardware at memory access speeds,” *Proc. INFOCOM*, IEEE Press, Piscataway, N.J., 1998, pp. 1240-1247.

[3] H. Liu, “Routing table compaction in ternary CAM,” *IEEE Micro*, Vol. 22, No.1, Jan.-Feb. 2002, pp.58-64.

[4] H. Nakahara, T. Sasao and M. Matsuura, “A CAM emulator using look-up table cascades,” *14th Reconfigurable Architectures Workshop*, RAW 2007, March 2007, Long Beach California, USA. CD-ROM RAW-9-paper-2.

[5] 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).

[6] G. Papadopoulos and D. Pnevmatikatos, “Hashing + Memory = Low Cost, Exact Pattern Matching,” in 15th. *International Conference on Field Programmable Logic and Applications*, Aug. 2005, pp. 39-44.

[7] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712-727, March 2006.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] T. Sasao, T. Nakamura, and M. Matsuura, “Representation of incompletely specified index generation functions using minimal number of compound variables,” *DSD-2009*, Aug. 2009, pp.765-772.

[14] I. Sourdis and D. Pnevatikatos, “Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system,” *FPL2003*, Lecture Notes in Computer Science 2778.

[15] D. E. Taylor, “Survey and taxonomy of packet classification techniques,” *ACM Computing Surveys*, Vol. 37, Issue 3, pp. 238-275, September, 2005.

[16] F. Yu, R. H. Katz, and T. V. Lakshman, “Gigabit rate packet pattern-matching using TCAM,” *Int. Conf. on Network Protocols* (ICNP2004), pp.174-183, 2004.