KUAI-EXACT: A NEW APPROACH FOR MULTI-VALUED LOGIC MINIMIZATION IN VLSI SYNTHESIS

Marek A. Perkowski, Pan Wu, Keith A. Pirkl

Department of Electrical Engineering, Portland State University
Portland, Oregon 97207, tel. (503)464-3806x23.

ABSTRACT

A new logic minimizer called KUAI-EXACT has been designed to generate the exact minimum solutions for multi-valued input logic expressions. The advantage of this minimizer is to generate as few prime implicants as possible. A new algorithm is presented here for directly generating essential prime implicants in a time close to that for generating a prime implicant by the ESPRESSO-MV "expansion" process. We discuss how to generate the secondary essential primes implicants in order to avoid setting up a covering table, and present the corresponding algorithms for non-cyclic functions. We also discuss the case in which a covering table should be created for obtaining an exact minimum solution and consider how to use the parallel processing techniques for the best speedup.

1. INTRODUCTION

After the "Expansion" process was first created by Hong [1] in 1974, it was widely used in many well known minimizers for logic minimization, usually with some minor improvements, for example, McBOOLE [2], MINI-II [3], ESPRESSO-II,-MV,-EXACT [4],[5],[6]. The reason for this is that it is considered to be the fastest process for generating prime implicants which are then used to generate the essential prime implicants and set up the covering table. Recently, therefore, with the number of input variables in VLSI circuits increasing, this process becomes more and more time consuming. This is because the number of all prime implicants can be of the order of $3^n/n$ for two-valued input n-variable functions [1] and will be even larger for multi-valued input functions [3]. Therefore, it is urgent to design a new algorithm for directly generating essential prime implicants. In this paper, the algorithm for the above task will be presented and its time for generating an essential prime implicant is close to that of generating a prime implicant by the ESPRESSO-II "expansion" process.

The other reason for generating all prime implicants is to set up a covering table which will help to obtain an exact minimum solution [7], [8], [9]. In ESPRESSO-II and ESPRESSO-MV, a covering table, as well as the generation of all prime implicants, is not needed because only the near minimum solutions are generated. Usually, these solutions are satisfactory for the design requirements but sometimes they are not. Also, it is hard to predict how far these solutions are from the minimal solutions. McBOOLE and ESPRESSO-EXACT, two recent typical algorithms for exact minimization, do not overcome the above problem of generating all prime implicants. In this paper, we avoid using a covering table, using the following principle: "for a non-cyclic function, an exact minimum solution is formed by only the essential and secondary essential prime implicants of the function". The algorithm will also be presented for generating the secondary essential prime implicants without a covering table.

For a cyclic function, we still need a covering table to aid in obtaining an exact minimum solution [10]. After, however, deleting all essential and secondary essential prime implicants from the original function, the reduced function will contain fewer cubes in the ON-set and more cubes in DC-set (don't-care set). Therefore, the number of all prime implicants for the reduced function is much less than that for the original function. Also, to the above reduced function with many don't-care, we propose a new algorithm to generate all prime implicants much faster than the "expansion" process of ESPRESSO-II. At the end of the paper, a parallel algorithm is presented for solving the covering problems. This algorithm is especially very efficient for functions with many symmetries in cycles.

In recent years, a multi-valued input logic expression is often employed as input data to logic synthesis instead of a binary-valued input expression [11]-[15]. The reason is that the silicon area required for the realization of multi-valued input logic (PLA with input decoders) is smaller than one required for two-valued input logic in PLA. Also, the multiple-output minimization for PLA optimization can be manipulated as a special case of a multi-valued minimization. A detailed discussion of multi-valued input logic is presented in [17], [5], [16]. In this paper, all considerations will be applied to the multi-valued input logic functions. Also, the approach presented here was especially created to find *exact* solutions with very good speed-ups on parallel computers with many processors. The report [17] detailizes parallelization.

2. THE DESCRIPTION OF KUAI-EXACT

KUAI-EXACT, where "KUAI" is translated from Chinese by pronunciation and means "fast", is an algorithm for multi-valued logic minimization. It can be applied to both completely and incompletely specified logic functions. It can accept either the ON-set and OFF-set or the ON-set and DC-set as input data. The sequence of processes of KUAI-EXACT is described below:

a) *Subminimal implicants*; This process generates the subminimal implicants from the initial ON-set of the function. The purpose of using the subminimal implicants instead of minterms as the input data of the next process is to save the execution time. Also, the purpose of using them instead of initial ON-cubes is to guarantee that all prime implicants can be generated which means obtaining an exact minimum solution.

b) *Essential primes*; Use a fast and efficient method to generate all essential prime implicants without generating all prime implicants. The time for an essential prime implicant generation is close to the time for generating a prime implicant by the "expansion" process in ESPRESSO-MV.

c) *Secondary essential primes*; Generate all secondary essential prime implicants to obtain an exact minimum solution without solving the covering problem for non-cyclic functions. This procedure is only applied to the reduced function which contains all non-essential primes of the original function.

d) *Extension*; Use a fast and efficient algorithm to generate all prime implicants for the remaining cyclic function. This process is totally different from the "expansion" process of ESPRESSO-MV in its principle.

e) *Covering search*; Solve the covering problem by the parallel search method.

I. Subminimal implicants

Definition 1. The *minimal* implicant of an ON-cube c, denoted by MI(c), is defined as the product of all prime implicants covering the cube c.

Definition 2. The *subminimal* implicant, denoted by SI, is defined as any implicant which is contained in a minimal implicant.

Theorem 1. Starting from subminimal cubes, there is a guarantee for generating all prime implicants. In the case of starting from arbitrary initial cubes, there is no guarantee to generate all prime implicants. (All proofs in [17]).

Algorithm 1. {To find all subminimal cubes for function F}

```
(1) Find the set CONS(ON) of all consensuses of cubes from the initial cover of the function;
(2) Find the set PROD(F) of all products of pairs of cubes from an
    CONS(ON) and an ON(F):
    $PROD(F) = \{ c_i \cap c_j \mid c_i \in ON(F) \; \& \; c_j \in CONS(ON) \}$;
(3) Find the set SI(F) of all subminimal cubes by:
    $SI(F) = (ON(F) \# CONS(ON)) \cup PROD$;
```

Example 1. Consider function f = $bd + \bar{a}\bar{b}\bar{c}d$ to generate all subminimal cubes, step by step, by using Algorithm 1. To ON(f) = ($\bar{a}\bar{b}\bar{c}d$, bd), (1) CONS(ON) = {$\bar{a}d\bar{c}$}, (2) PROD(F) = {$\bar{a}\bar{b}\bar{c}d$, $\bar{a}b\bar{c}d$}, and (3) SI(F) = {$\bar{a}\bar{b}\bar{c}d$, $\bar{a}b\bar{c}d$, abd, bcd}. The corresponding ON(F), CONS(ON), PROD(F), and SI(F) are shown in Fig. 1.

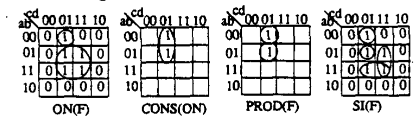


Figure 1. The sets for Example 1.

II. Essential Primes

Definition 3. The *supercube* SC of the set of cubes, ($c_1, c_2, \ldots\ldots, c_k$), is defined as a smallest cube that covers all the cubes in this set.

Definition 4. The *overexpanded* cube of a cube c, denoted OC(c), is defined as the supercube of all prime cubes which cover the cube c.

According to the Definition 4, a direct and efficient method, for generating the overexpanded cube OC(c) of a cube c, is to first generate all prime cubes which cover the cube c. Then OC(c) will be the bit-by-bit OR of these prime cubes. Although this method is proper, it would take much time to generate all prime cubes, and would need much memory to store them. A new method, which will avoid generating all prime cubes, will be discussed below.

Definition 5. A *single-variable-expansion* cube (SVE cube, for short) of cube c is defined as a cube which is obtained by expanding only one variable of the cube c until no more expansion can be performed. Thus, if c = $X_1^{S_1}X_2^{S_2}\ldots\ldots X_n^{S_n}$, the expression of single-variable-expansion on variable *i* is as follows:
$SVE\,(c,X_i) = X_1^{S_1}\ldots X_i^{R_i}\ldots X_n^{S_n}$; Where R_i is the maximal possible set for X_i.

Theorem 2. The overexpanded cube OC of a cube c can be generated by bit-by-bit OR of all single-variable-expansion cubes of the cube c.

From the Definition 5, it is obvious that a SVE(c) cube may not be a prime cube. Then, by using the above theorem, to generate all prime cubes is not necessary for generating an overexpanded cube.

In the expansion process of ESPRESSO-MV, a cube is expanded to a prime cube by expanding all variables contained in the cube c. So, if there are m variables in a cube c, the time, for expanding the cube c to a prime cube, will be m times of the time for the single-variable-expansion. Here, the time for generating an overexpanded cube is m times of the time for getting SVE(c) cubes plus the time for one union operation, and, therefore, it is almost the same as the time for a prime implicant expansion.

Algorithm 2. {To generate the overexpanded cube of a cube c}
(1) find all signal-variable-expansion cubes by the "expansion" process.
(2) find the overexpanded cube by: $OC\,(c) = \bigcup_{i=1}^{i=m} SVE\,(c,X_i)$;

Example 2. For a function with an ON-set as {$X^{(3,4)}Y^{(1,2,3,4)}$, $X^{(2,3,4)}Y^{(2,3,4)}$}. Consider a subminimal cube c = $X^4Y^{(2,3)}$, then, SVE(c, X) = $X^4Y^{(1,2,3,4)}$, SVE(c, Y) = $X^{(2,3,4)}Y^{(2,3)}$, and overexpanded cube OC(c) = SVE(c, X) ∪ SVE(c, Y) = $X^{(2,3,4)}Y^{(1,2,3,4)}$.

All of the subminimal cubes in SI(F) are listed one by one from top to bottom. When the essential primes process is executed, the subminimal cube on the top of the list is taken first, then the next. For setting up this kind of list, the set SI(F) is divided into three subsets. One is the set PROD(F) which contains the products of the initial cubes and consensus cubes, and is obtained by employing the step 2 of Algorithm 1. Another is the set REST(F) which is equal to ON(F) # CONS(ON), and is obtained by using the step 3 of the Algorithm 1. The last one is the set SEPC(F) which is empty before the process of essential prime implicants is executed, and will contain those subminimal cubes that are adjacent to essential prime cubes.

The ordering of these three subsets is that the set PROD(F) is on the top of the list, then the set SEPC(F), and last is the set REST(F). Also inside the sets PROD(F) and SEPC(F), the smaller cubes are put on the top of the larger cubes. The reason for this kind of ordering is to tend to put those cubes, that are hard to merge with others or, in other words, contain some essential vertices [1], on the top of the list. This will result in first generating those overexpanded cubes that have higher possibility to be essential prime cubes. After an essential prime cube is found, all subminimal cubes covered by this essential prime cube will be deleted. Therefore, the number of iterations can be reduced. In the set REST(F), the larger cubes are put on the top of the smaller cubes. The reason here is that, if an essential prime cube is found, this essential prime cube is expected to cover as many subminimal cubes as possible. This would also reduce the number of iterations.

Theorem 3. To a given function F with a subminimal cover SI(F), the overexpanded cube OC(c) of a subminimal cube c is an essential prime cube, if it satisfies the condition: OC(c) ⊆ SI(F).

Algorithm 3. {To generate all essential prime cubes}.
(1) Take a subminimal cube c_s from the ordering list of SI(F), and find the overexpanded cube OC(c_s) by using the Algorithm 2.
(2) If OC(c_s) ⊆ SI(F), then;
  Sharp OC(c_s) from SI(F), replace it with don't-cares in SI(F), and put it to the final solution set .
  Find all subminimal cubes which are adjacent to OC(c_s) in the set PROD(F) ∪ REST(F), and move them into the set SEPC(F).
(3) Iterate the above steps until all subminimal cubes in the set PROD(F) ∪ REST(F) have considered.

Example 3. Consider the function $f = X^0Y^1 + X^{(1,3)}Y^{(1,3)}$. The corresponding subminimal set is {$X^0Y^1, X^{(1,3)}Y^1, X^{(1,3)}Y^3$}, the set PROD($f$) is {$X^0Y^1$, $X^{(1,3)}Y^1$}, and the set ON(f)#CONS(f) is {$X^{(1,3)}Y^3$}. By using the above rules for cubes ordering, the ordering list of SI(f) is {X^0Y^1, $X^{(1,3)}Y^1$, $X^{(1,3)}Y^3$}. And the corresponding three ordered sets are:

PROD(f) = {$X^0Y^1, X^{(1,3)}Y^1$}, SEPC(f) = ∅, and REST(f) = {$X^{(1,3)}Y^3$}.

The sequence of steps of applying Algorithm 3 is presented below.
(1) Take the cube (X^0Y^1) first and obtain OC(X^0Y^1) = $X^{(0,1,3)}Y^1$.
(2) Since OC(X^0Y^1) ⊆ SI(f), the cube $X^{(0,1,3)}Y^1$ is an essential prime cube. Then this cube is deleted from the set SI(f) and put in the final solution. The cube $X^{(1,3)}Y^3$, which is adjacent to the above OC, is put in the set SEPC(f). Now the status for the three sets is:

PROD(f) = ∅, SEPC(f) = {$X^{(1,3)}Y^3$}, and REST(f) = ∅.

(3) Take the cube $X^{(1,3)}Y^3$, the overexpanded cube is $OC = X^{(1,3)}Y^{(1,3)}$.
(4) Since the cube OC = $X^{(1,3)}Y^{(1,3)}$ is also an essential prime cube, this cube is deleted from the subset of SI(f) and put in the final solution.
(5) Stop here, because the set of SI(f) is empty.

III. Secondary essential primes

After all essential primes has been extracted from the set of SI(F) of the given function F, the cubes remaining in SI(F) will cover a new function F_1. Obviously, the function F_1 is included in the function F and contains only the secondary essential primes and the cyclic cubes. In this section, it is discussed how to find all the secondary essential primes in the set SI(F_1).

Definition 6. A *redundant prime implicant,* denoted e_r, is defined as a prime implicant which contains only the subminimal cubes included in other prime implicants and possible don't-cares.

Definition 7. A *secondary essential prime implicant,* denoted e_s, is defined as a prime implicant which will become an essential prime implicant after all redundant prime implicants which intersect with e_s are deleted.

A secondary essential prime is actually not an essential prime. It will become the essential prime only after the essential primes, which are adjacent to it, are first all deleted. Therefore, secondary essential primes are always adjacent to the essential primes.

After all essential primes are extracted from the function, the sets PROD(F) and SEPC(F) are merged into one set denoted by SEPC(F_1). The set REST(F) now becomes the set REST(F_1). The ordering for SEPC(F_1) is that the set SEPC(F) is on the top of the set PROD(F). The reason is that the cubes in the set SEPC(F) have more possibility to be the secondary essential primes than the cubes in the set PROD(F) because they are all adjacent to the essential primes. In the set PROD(F), only those cubes which are actually the essential primes but are surrounded by some don't-cares will be the secondary essential primes. The inside ordering of the sets SEPC(F) and PROD(F) is kept the same as before because the property of the secondary essential primes is the same as that of the essential primes. The secondary essential primes found are still expected to cover more subminimal cubes.

Only the set SEPC(F_1) is ordered and considered for generating all secondary essential primes and the set REST(F_1) will be ignored. The reason is that no cubes which are adjacent to essential prime cubes will exist in the set REST(F_1). All cubes which are adjacent to the essential primes have been put into set SEPC(F) in Algorithm 3 already. The set REST(F_1) is only considered at the time when the set SI(F_1) = SEPC(F_1) ∪ REST(F_1) is considered. After a secondary essential prime is found, all cubes that are adjacent to this secondary prime and are included in SI(F_1) will be appended to the bottom of the list of the ordered set SEPC(F_1).

From Definitions 6 and 7, a direct and efficient method to generate the secondary essential implicants from the subminimal cubes is to generate the set PI(c_s) of all prime cubes which cover the specified subminimal cube c_s. Then, check which prime cubes in PI(c_s) are the redundant prime cubes and delete them from PI(c_s). If there exist two or more remaining prime cubes in PI(c_s), no secondary essential primes will exist in PI(c_s). If there is only one prime cube left in PI(c_s), this prime cube is a secondary essential prime.

Algorithm 4. {To generate all secondary essential prime cubes}.

(1) Take first subminimal cube c_s from the ordered set SEPC(F_1) and generate all prime cubes PI(c_s) which all cover c_s.

(2.1) Take the largest prime cube P_i from PI(c_s), and sharp c_s from P_i. If $(P_i \# c_s) \subseteq$ DC(F_1), then, sharp P_i from the set PI(c_s).

(2.2) Take the current largest prime cube, and iterate the step 2.1) until one of two following cases will happen.

- a) If there exist two prime cubes which have been checked not to be the redundant prime cubes, then go to step 3).
- b) If there is only one prime cube left in PI(c_s), move it in the final solution. Find all subminimal cubes, which are adjacent to P_i, in the SI(F_1), and append them to the bottom of the list SEPC(F_1).

(3) Take another subminimal cube from the current top of SEPC(F_1), iterate step 2 until the set SEPC(F_1) is empty or all subminimal cubes in the set are checked not to be the secondary essential primes.

Example 4. Consider a function shown in Fig. 2.a. After all subminimal cubes are generated, the set PROD(F) contains all subminimal cubes and both REST(F) and SEPC(F) are empty.

After the essential cube $ab\bar{e}$ is found and deleted from the set SI(F), the new K-map is shown in Fig. 2.b. Now the set SEPC(F_1) contains the cubes t_1, t_2, and t_3 shown in Fig. 2.c. The PROD(F_1) becomes the set [SI(F) # $ab\bar{e}$ # t_i], where i=(1,2,3), and REST(F_1) is still empty.

Take the cube t_1, and generate all prime cubes which cover t_1: $c_1 = \bar{a}\bar{b}\bar{c}\bar{d}$, $c_2 = \bar{b}\bar{c}\bar{d}\bar{e}$. Since the cube c_2 is a redundant prime cube and the cube c_1 is not, c_1 is a secondary essential cube. It is put in the final solution and the cube $\bar{a}b\bar{c}de$, which is adjacent to c_1, is appended to the bottom of the set SEPC(F_1). And, similarly, the cube t_2 is a secondary essential cube and the cube $\bar{a}b\bar{c}de$ is appended to the set SEPC(F_1).

To t_3, all prime cubes are $p_1 = a\bar{b}cd$, $p_2 = acde$, and $p_3 = \bar{b}cde$. Since p_1 is redundant cube and both p_2 and p_3 are not the redundant cubes, t_3 is not a secondary essential cube.

Repeat the above procedure for the new SEPC, the final results are shown in Fig. 2.d. There exists a covering cycle.

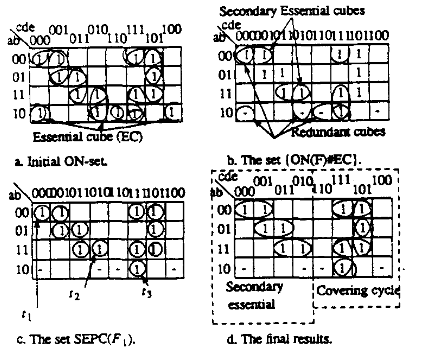


Figure 2. The procedures for Example 4.

IV. Extension

After the processes described in last two sections are executed, all essential and secondary essential primes have been extracted. If the set SI(F) is empty, KUAI-EXACT will stop here and the final solution contains all essential and secondary essential primes. If the SI(F) is not empty, it will contain only the cyclic cubes which cover a new function, denoted by f. To find an optimal solution for the function f, a covering table should be used. Therefore, all prime implicants should be generated.

All cubes left in the sets SEPC(F_1) and REST(F_1) are now merged into the set CC(f). The list of CC(f) is sorted so that the largest cube is put on the top of it, and the smallest one is at the bottom. The reason is that the largest cube may covered by more prime cubes or by larger prime cubes. These larger prime cubes will have more possibility to completely cover other prime cubes, which will then be generated from other subminimal cubes. Then the covered (completely) prime cubes are removed before they are put in the covering table. This will reduce the numbers of rows in the covering table.

Definition 8. For a multi-valued input Boolean function, the *associate primes* of c_1 are defined as those prime cubes which cover one specific subminimal cube c_1. An associate cover of c_1 is the set of all associate primes of c_1.

From the current top of the ordering list of CC(f), a subminimal cube c_s is taken. All associate primes of c_s will be generated and will be compared to the prime cubes which have been put in the covering table already. If any prime cube c_p in PI(c_s) is the same as or is contained in other prime cube in the covering table, c_p is deleted. The remaining cubes from PI(c_s) are added to the covering table where the subminimal cube will be added as a new column and the prime cube will be added as a new row. Also, some ones are entered to represent the relationship between the new rows and all the columns. This procedure will be repeated until the set CC(f) becomes empty.

Definition 9. Consider two cubes c_1 and c_2, where $c_1 = x_1^{S_1} x_2^{S_2} \dots x_n^{S_n}$ and $c_2 = x_1^{R_1} x_2^{R_2} \dots x_n^{R_n}$. An *Asymmetrical Union* of c_1 and c_2 is defined as:

$$asum(c_1, c_2) = \begin{cases} \varnothing & \text{if } S_i \cap R_i \neq S_i, \ (i=1,2\dots,n) \\ c_2 & S_i \subset R_i \end{cases}$$

If $\vec{e}$ is an array of cubes, then, $asum(c_1, \vec{e}) = \bigcup_{e_i \subset \vec{e}} asum(c_1, e_i)$.

Theorem 4. Assume that an n-variable multi-valued input Boolean function f contains m cubes, t_i (i = 1, 2, ... m), in the OFF-set. An associate cover H(c_1), where $c_1 \in$ SI(f), can be generated by: $H(c_1) = \bigcap_{i=1}^{m} asum(c_1, \bar{t}_i)$

When an associate cover is generated by the above Theorem, some redundant cubes are generated as well, particularly, for multi-valued logic. To reduce the number of those redundant cubes, two considerations are discussed. First, the cubes in the OFF-set should be rearranged at the beginning of the process "extension" so that the number of the cubes in the OFF-set could be decreased as much as possible and the sizes of the cubes in the OFF-set could be as large as possible. A detailed discussion and corresponding algorithms are presented in [17]. Second, if two or more cubes of the OFF-set overlap, the associate cover would contain many redundant cubes. To reduce this kind of redundancy, we choose the largest OFF-cube to do the Asymmetrical Union with the subminimal cube. Then a Sharp operator is employed to cancel all other cubes from the result of the Asymmetrical Union. Because the number of the overlapped cubes is small, a Sharp operator will not take too much time here.

Algorithm 5. {To generate an associate cover for a subminimal cube}

(1) Take the first subminimal cube c_l from the ordered set CC(f) and compare every part of c_l to the corresponding part of all the OFF-cubes.

(2) To the OFF-cubes which have one or more parts covering the corresponding parts of c_l, take them to the Asymmetrical Union with c_l.

(3) To the rest of OFF-cubes, find the cubes which are disjoint with all of the other OFF-cubes. Take them to the Asymmetrical Union with c_l.

(4) In the remaining OFF-set, find the disjoint groups and do:

- a) Find the largest cube in the each group, and compute Asymmetrical Union of this cube and c_l. If no single largest cube exists, choose any of them.
- b) Sharp all of the other OFF-cubes in the group from the result of the above step.

(5) Compute the Intersection of the results in the steps 2), 3), and 4.b).

Example 5. Consider a multi-valued function with an OFF-set as follows:

$$OFF(F) = \begin{bmatrix} S_{1_1}—S_{2_1} \\ S_{1_2}—S_{2_2} \\ S_{1_3}—S_{2_3} \\ S_{1_4}—S_{2_4} \end{bmatrix} = \begin{array}{c} \begin{array}{cc} X_1 & X_2 \end{array} \\ \begin{bmatrix} 1100—00010000 \\ 0100—00110000 \\ 0010—00000110 \\ 0011—00000100 \end{bmatrix} \end{array} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{bmatrix}$$

And, the ON-cube: $x_1 = (R_1—R_2) = (1100\text{-}10000000)$,
To t_1, because $R_1 \subset S_{1_1}$, therefore:

$$\bar{t}_1 = \begin{bmatrix} 0011—11111111 \\ 1111—11101111 \end{bmatrix} \begin{array}{l} \dots\dots\bar{t}_{1_1} \\ \dots\dots\bar{t}_{1_2} \end{array}$$

and $asum(x_1, \bar{t}_1) = \bar{t}_{1_2} = (1111-11101111)$.

To t_2, because $R_1 \not\subset S_{1_2}$, $R_2 \not\subset S_{2_2}$, and the cube t_2 is disjoint to the cubes t_3 and t_4, therefore;

$$\bar{t}_2 = \begin{bmatrix} 1011-11111111 \\ 1111-11001111 \end{bmatrix} \begin{matrix} \ldots\ldots\bar{t}_{2_1} \\ \ldots\ldots\bar{t}_{2_2} \end{matrix}$$

and $asum(x_1, \bar{t}_2) = \bar{t}_{2_2} = (1111-11001111)$.

To t_3, because $R_1 \not\subset S_{1_3}$, $R_2 \not\subset S_{2_3}$, but $S_{1_3} \subset S_{1_4}$, which means the cubes t_3 and t_4 are overlapped. Then it is not necessary to compute the asymmetrical union for both the cube t_3 and cube t_4. Since the cube t_3 is not larger than the cube t_4 and vice versa, the cube t_3 is chosen randomly,

$$\bar{t}_3 = \begin{bmatrix} 1101-11111111 \\ 1111-11111001 \end{bmatrix}$$

and $asum(x_1, \bar{t}_3) = \bar{t}_3$.

Sharp the cube t_4 from the set $asum(x_1, \bar{t}_3)$.

$$(asum(x_1, \bar{t}_3)) \# t_4 = \begin{bmatrix} 1100-11111111 \\ 1101-11111011 \\ 1111-11111001 \end{bmatrix}$$

At last, compute the intersection for final result:

$$H(x_1) = (asum(x_1, \bar{t}_1)) \cap (asum(x_1, \bar{t}_2)) \cap (asum(x_1, \bar{t}_3) \# (t_4))$$

$$= \begin{bmatrix} 1100-11001111 \\ 1101-11001011 \\ 1111-11001001 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

V. Covering search

The tree search methods are the good approaches for solving a covering table, but they are usually time consuming. We are interested in using parallel processing for the design of a fast search method. The *exhaustive technique*, to search every possible combination of the problem, is easily improved for the above reason. Other parallel algorithm in [18].

In the set covering problem, from a given covering array and costs of rows, the rows must be selected to cover all the columns and minimize the total cost of selected rows. Here is an example of the covering array in the form of a list.

```
                             1 2 3 4 5 6 7 8 9
A. 1 3 5 7                 A. 1 0 1 0 1 0 1 0 0
B. 4 6 9 7                 B. 0 0 0 1 0 1 1 0 1
C. 8 3 2         <==>      C. 0 1 1 0 0 0 0 1 0
D. 6 1 4 5 7 8             D. 1 0 0 1 1 1 1 1 0
E. 9 3 5                   E. 0 0 1 0 1 0 0 0 1
```

A take/don't take parallel method is represented by a binary string which means using a five digit binary number to represent all possible combinations of rows as candidates for a solution. For example, if we take the string 00101 and the string 11010, the selection processes would look like this:

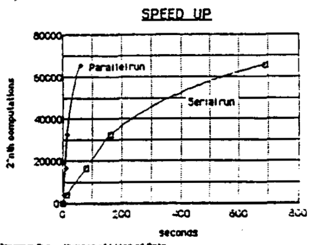
```
     1 2 3 4 5 6 7 8 9            1 2 3 4 5 6 7 8 9
A    Dont take-------             1 0 1 0 1 0 1 0 0
B.   Dont take-------             0 1 1 0 0 0 0 0 0
C.   0 1 1 0 0 0 0 1 0  .         Dont take-------
D.   Dont take-------             1 0 0 1 1 1 1 1 0
E.   0 0 1 0 1 0 0 0 1            Dont take-------
     (String 00101)               (String 11010)
```

The strings describe candidates--each 1 means the selected row, and the cost of the string is the number of ones. So, it is possible to test the string by Boolean (bit by bit) addition of the rows selected (marked 1) in this string to verify whether it is a complete solution. If it is a solution, its cost is calculated. If the new solution costs more than some solution already found (has more ones in the string), then it is discarded. The actual minimal cost is a global variable known to all processors. If we try to obtain the exact minimum solution, the set of all strings must be generated.

Let us observe that this method does not require time consuming and complicated array copying, but instead replaces it with fast binary word operations that would be even faster on special vector processors. A simplified pseudo-code for the parallel algorithm is:

```
PARALLEL
Do a count from myid to $2^{\#rows}$ - 1 step (number of processors working)
  clear tempsolution   : storage of array
  test [each bit of count]
    { if bit set
      [ add row [indexed by bit] to tempsolution ] }
    test [ tempsolution for complete coverage ]
    [ if coverage
      [ test [ if solution is better than best ]
        { set best to better than best
          set goodsolution to count : count was an index } } }
  endcount
```

The program was written in C for Sequent *Balance*™ computer, and runs both as serial and as parallel. The speed-up 10.2 was achieved on 11 processors (see Fig. 3).



Program Run	Number of Lines of Data				
	7	12	14	15	16
Serial	.99	14.98	79.06	160.57	609.59
Parallel	1.03	2.96	9.26	15.43	59.78

Figure 3. Sample output time seconds.

3. LITERATURE

[1] S.J. Hong, R.G. Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. R&D*, Vol.18, pp.443 - 458, Sept. 1974.

[2] M.R. Dagenais, V.K. Agarwal and N.C. Rumin, "McBOOLE: A new procedure for exact logic minimization," *IEEE TCAD*, Vol.CAD-5, No.1, pp.229-237, Jan. 1986.

[3] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE TC*. Vol.33, No.10, pp.879-894, Oct. 1984.

[4] R.K. Brayton, G. D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Kluwer Academic Publishers, 1984.

[5] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE TCAD*, Vol.CAD-6, No.5, pp.727-749, Sept. 1987.

[6] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Exact minimization of multiple-valued functions for PLA optimization," *Proc. ICCAD*, Nov. 1986.

[7] W.V. Quine, "The problem of simplifying truth functions," *Amer. Math. Monthly*, Vol.59, p.521, 1952.

[8] W.V. Quine, "A way to simplify truth functions," *Amer. Math. Monthly*, Vol.62, p.627, 1955.

[9] E. J. McCluskey, Jr., "Minimization of boolean functions," *Bell System Tech. J.*, Vol.35, pp.1417-1443, Apr. 1956.

[10] M.A. Perkowski, J. Liu and J. Brown, "Quick Software Prototyping: CAD Design of Digital CAD Algorithms," in *"Progress in Computer Aided VLSI Design,"* edited by G. Zobrist. Ablex Publishing Corp., 1989 (in print).

[11] H. Fleisher and L.I. Maissel, "An introduction to array logic," *IBM J. of R&D*, Vol.19, pp.98-109, Mar. 1975.

[12] T. Sasao, "An application of multiple-valued logic to a design of Programmable Logic Arrays," *Proc. 8th ISMVL*, 1978.

[13] Y.S. Kuo, "Generating essential primes for a boolean function with multiple-valued inputs," *IEEE TC*. Vol.36, pp.356-359, Mar. 1987.

[14] T. Sasao, "MACDAS: Multi-level AND-OR circuit synthesis using two-variable function generators," *Proc. 23rd DAC*, pp.86-93, Jun. 1986.

[15] T. Sasao, "An algorithm to derive the complement of a binary function with multiple-valued inputs," *IEEE TC*. Vol.34, pp.131-140, Feb. 1985.

[16] S.Y.H. Su and P.T. Cheung, "Computer minimization of multivalued switching functions," *IEEE TC*. Vol.21, No.9, pp.995-1003, Sept. 1972.

[17] M.A. Perkowski, P. Wu, "A new approach to exact minimization of Boolean functions with multi-valued inputs," Technical Report, Dept. E.E., Portland State Univ., Portland, Oregon, Sept. 1988.

[18] K.A. Pirkl, "Comparison of parallel set covering algorithms," Report, P.S.U. 1989.