Overlapping Functional Decomposition in Logic Synthesis

Student: Li-Wen Hung
Advisor: TingTing Hwang

Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan 30043

June,30,2000

Contents

List of Figures

List of Tables

Abstract

Functional decomposition is an effective technique to restructure logic networks. Previous researches have studied how to select variables in bound set, how to encode compatible classes, etc. However, most of previous works consider non-overlapping structures [2, 4, 8, 9, 15]. Circuit structures with overlapping variable are more flexible and may have lower area costs than the non-overlapping structures in some cases [1]. Therefore, in this paper, we will study nondisjunctive functional decompositions for Boolean functions.

We first propose an algorithm to select overlapping variables from the bound set of a function. After the overlapping variable selection algorithm is applied, the compatible classes are paired to assign don't cares. The objectives of the don't care assignment heuristic are to reduce the compatible classes and to increase the decomposability of the image function that will be decomposed in the next level. Finally, we proposed an algorithm to improve the compatible class encoding algorithm proposed in HYDE [2]. Then, we apply this nondisjunctive functional decomposition technique to look-up table (LUT) based FPGA synthesis. The experimental results show that our overlapping approach can lead to better mapping results on some particular cases of benchmarks.

Chapter 1

Introduction

Multilevel logic synthesis is performed in two phases: technology independent optimization and technology mapping. In the first phase, Boolean network is re-structured and optimized by functional transformations. In the second phase, the results of the first phase are mapped to library cells which are defined by such technologies as standard cells, FPGA's, etc..

Functional transformation (network re-structuring) techniques include algebraic decomposition and functional decomposition. In algebraic decomposition, common subexpression identification is used to restructure a logic network. In common subexpression identification, finding *divisor* of an expression is a key operation (We say that p is a divisor of function f if $f = p \cdot q + r$ and $p \cdot q \neq 0$). In functional decomposition, a function $f(X)$ is transformed to $g(\alpha(X_1), X_2)$ where $X_1 \cup X_2 = X$. By this approach, how to select X_1 from X so that the decomposed function is minimized is one of the key steps. It is clear that algebraic decomposition can be seen as a special case of functional decomposition where divisor p corresponds to the subfunction $\alpha(X_1)$. In this thesis, we will study functional decomposition to restructure a network.

Disjunctive functional decomposition first proposed by Ashenhurst [15] and Roth & Karp [4] is one of two types of functional decomposition. It has been adopted by many Look-Up-Table based (LUT-based) technology mappers [1, 2, 6, 16] which have

produced good results. However, disjunctive functional decomposition requires that function f be decomposed so that $f(X) = g(\alpha(X_1), X_2)$ where $X_1 \cup X_2 = X$ and $X_1 \cap X_2 = \emptyset$. This constraint of non-overlapping variable partition is too restrictive for some functions to result in satisfactory decomposition. Therefore, in this thesis, we will relax this constraint and discuss functional decomposition for overlapping variable partitions.

First, we will propose an algorithm to select variables that are overlapping between the two partitioned input sets for nondisjunctive functional decomposition. After the overlapping variable selection algorithm is applied, the don't care sets produced by variable overlapping can be utilized to reduce the complexity of subfunctions. We will then propose an algorithm to encode subfunctions so that the decomposability of resultant functions can be improved. Finally, we will apply our nondisjunctive functional decomposition technique to technology mapping for LUT based FPGA.

The remainder of this thesis is organized as follows. In Chapter 2, we review previous work on functional decomposition and give definitions and terminologies. In Chapter 3, the overlapping variable selection algorithm and compatible classes pairing strategy for don't care assignment are discussed. In Chapter 4, we improve the HYDE algorithm [2] for the compatible class encoding problem. Then, we utilize our proposed method to perform technology mapping for LUT-based FPGA synthesis. Chapter 5 shows the experimental results and concluding remarks will be given in Chapter 6.

Chapter 2

Functional Decomposition

In Section 2.1, we present the definitions of functional decomposition and terminologies. In Section 2.2, we will review the synthesis of disjunctive and nondisjunctive functional decompositions based on OBDD representation [1, 6].

2.1 Definition and Terminology

Functional decomposition of a function $f(x_1, ..., x_n)$ is defined as

$$f = g(\alpha_1(X^B), ..., \alpha_t(X^B), X^F) = g(\vec{\alpha}(X^B), X^F), \tag{2.1}$$

where X^B and X^F are sets of input variables and $X^B \cup X^F = \{x_1, ..., x_n\}$. The sets X^B and X^F are called the **bound set (λ set)** and the **free set (μ set)**, respectively. If $X^B \cap X^F = \emptyset$, this decomposition is called **disjunctive decomposition** . Otherwise, it is called **nondisjunctive decomposition** . These two forms of decomposition of f are illustrated in Figure 2.1. In this Figure, input variable set is partitioned to two sets X^B and X^F where $X^B = \{x_1, x_2, ..., x_k\}$ and $X^F = \{x_{k+1}, x_{k+2}, ..., x_n\}$ in Figure 2.1(b) while $X^B = \{x_1, ..., x_i, ..., x_j\}$ and $X^F = \{x_i, ..., x_j, ..., x_n\}$ in Figure 2.1(c). Note that in Figure 2.1(c) input set $X^B \cap X^F = \{x_i, ..., x_j\}$ is called overlapping variables with respect to this nondisjunctive decomposition.

With respect to a partition $X = (X^B, X^F)$, we say that x_1 and x_2 are in the same **compatible class** denoted as $x_1 \approx x_2$ if $x_1, x_2 \in X^B$ and $f(x_1, y) = f(x_2, y)$ for all $y \in X^F$. The minimum number of α function is *log(the number of compatible classes)*. The goal of functional decomposition is to find the minimal number of α functions and encode the α functions so that the α functions and g function (called image function) are minimal. However, it is very difficult to take both α functions and image function into consideration when encode α functions. In most of previous work [1, 2, 4, 8, 15, 16], functional decomposition is used in synthesizing LUT-based FPGA. Since the complexity of α function does not affect the quality of synthesis result, only minimizations of image functions are considered.

2.2 Synthesis of Disjunctive and Nondisjunctive Functional Decompositions

Functional decompositions have been implemented using Ordered Binary Decision Diagram (OBDD) [1, 6]. We illustrate the previous synthesis of α and image functions for disjunctive and nondisjunctive functional decompositions using OBDD. First, we give two definitions.

Definition 2.2.1 A Given OBDD with variable ordering $x_1 < ... < x_n$ representing $f(x_1, ..., x_n)$. Let $cut\_set(f, l)$ denotes the set of nodes whose level are greater than l and that have edges from nodes of level less than or equal to l. These nodes are the compatible classes of function f.

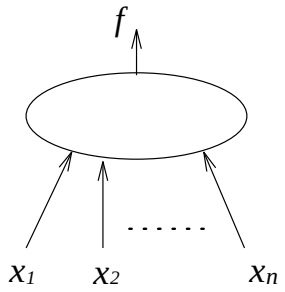
For an n-variable function f with a fixed variable order, if $|cut\_set(f, l)| \leq 2^t$, there exists a disjunctive decomposition as shown in Figure 2.1(b), where $X^B = \{X_1, ..., X_l\}$ and $X^F = \{X_{l+1}, ..., X_n\}$. $|cut\_set(f, l)|$ is the number of compatible classes, and t is the number of α functions.

Definition 2.2.2 A Given OBDD with variable ordering $x_1 < ... < x_n$ representing $f(x_1, ..., x_n)$. Let $s \leq l$ and $i \in \{0,1\}^{l-s+1}$. The $cut\_set\_nd(f, l, s, i)$ denotes $cut\_set(f_i, l)$, where f_i is the function resulting from assigning i to f at the variables from level s to level l.

For an n-variable function f with a fixed variable order, if $|cut\_set\_nd(f, l, s, i)| \leq 2^t$, there exists a nondisjunctive decomposition as shown in Figure 2.1(c), where $X^B = \{X_1, ..., X_l\}$ and $X^F = \{X_s, ..., X_n\}$. $|cut\_set\_nd(f, l, s, i)|$ is the number of compatible classes of function f_i, and t is the number of α functions of function f_i.

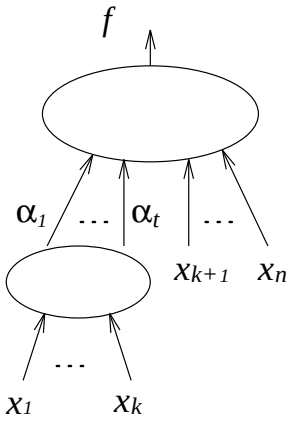
Take Figure 2.2 for example. Let the decomposition be disjunctive and $X^B = \{x_1, x_2, x_3\}$ and $X^F = \{x_4\}$. The $cut\_set(f, 3) = \{v_0, v_1, v_2\}$, which represents the set of compatible classes in Figure 2.2. Since the $| cut\_set(f, 3) | = 3 \leq 2^2$, we require two 3-input α functions. It can be described as $f = g(\alpha_1(x_1, x_2, x_3), \alpha_2(x_1, x_2, x_3), x_4)$. If we encode α functions as $(\alpha_1, \alpha_2) = (0,0)$, $(0,1)$, and $(1,0)$ to compatible classes v_0, v_1, and v_2 as in Figure 2.3(a), OBDD's representations of these α functions and image function are shown in Figure 2.3(b) and (c), respectively.

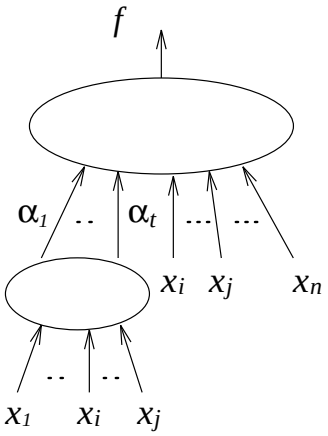
Now, consider overlapping decomposition. Suppose x_3 is selected to be overlapping variable. The $cut\_set\_nd(f, 3, 3, 0) = \{v_0, v_1\}$, and the $cut\_set\_nd(f, 3, 3, 1) = \{v_1, v_2\}$ in Figure 2.2. Because the $| cut\_set\_nd(f, 3, 3, 0) | = 2 \leq 2^1$ and $| cut\_set\_nd(f, 3, 3, 1) | = 2 \leq 2^1$, it only requires one 3-inputs α functions. The nondisjunctive decomposition form is $f = g(\alpha_1(x_1, x_2, x_3), x_3, x_4)$. Moreover, if we encode the compatible classes v_0 as 0 and v_1 as 1 when overlapping variable $x_3 = 0$, and encode v_1 as 0 and v_2 as 1 when $x_3 = 1$ in Figure 2.4(a), the OBDD representing α function and image function are showed in Figure 2.4(b) and (c).



$f(x_1,...,x_n)$

(a)





$X^B = \{x_1,...,x_k\}$

$X^F = \{x_{k+1},...,x_n\}$; $1<k<n$

(b)

$X^B = \{x_1,...,x_j\}$

$X^F = \{x_i,...,x_n\}$; $1<i<j<n$

(c)

Figure 2.1: Disjunctive and nondisjunctive forms of functional decomposition: (a) the original function (b) Disjunctive form (c) Nondisjunctive form

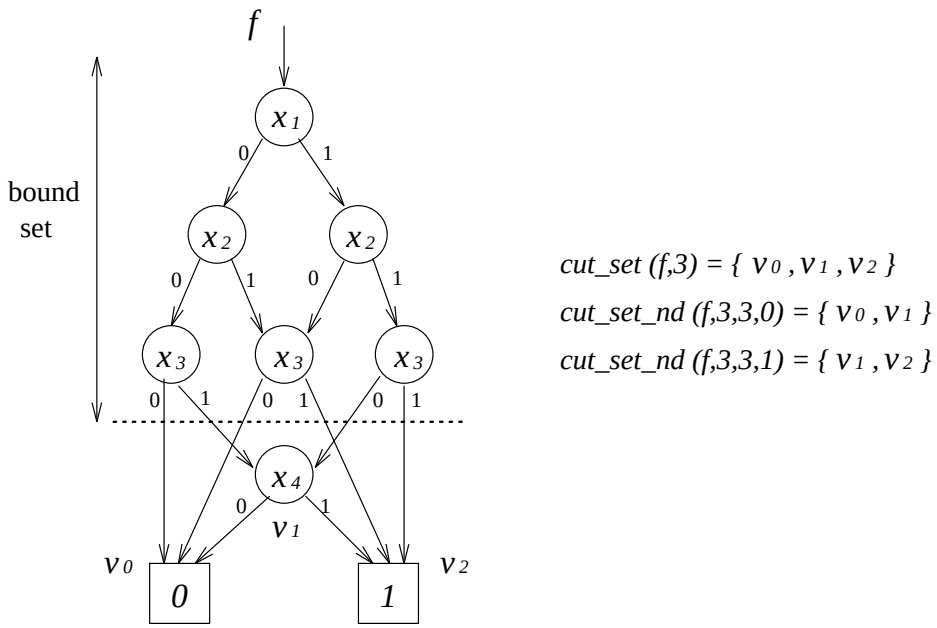
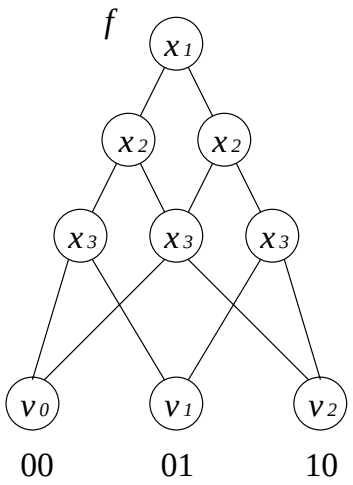
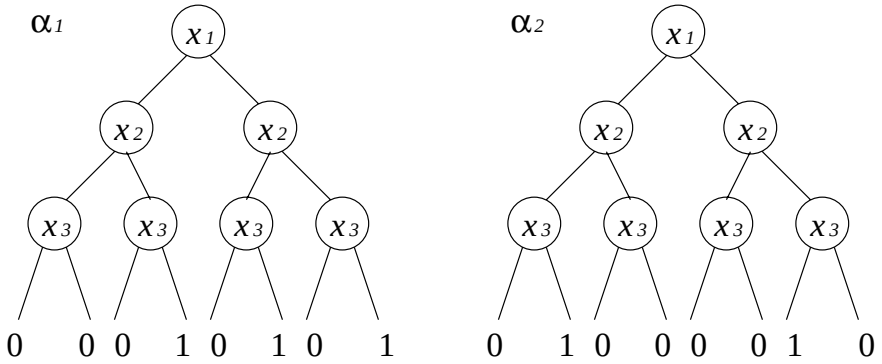


Figure 2.2: An OBDD and its cut_set information

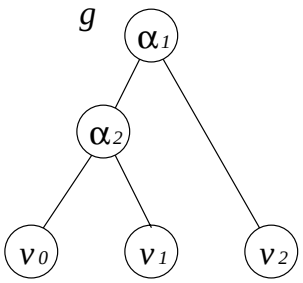


x_1	x_2	x_3		α_1	α_2
0	0	0	v_0	0	0
0	0	1	v_1	0	1
0	1	0	v_0	0	0
0	1	1	v_2	1	0
1	0	0	v_0	0	0
1	0	1	v_2	1	0
1	1	0	v_1	0	1
1	1	1	v_2	1	0

(a)



(b)



(c)

Figure 2.3: OBDD representing disjunctive decomposition

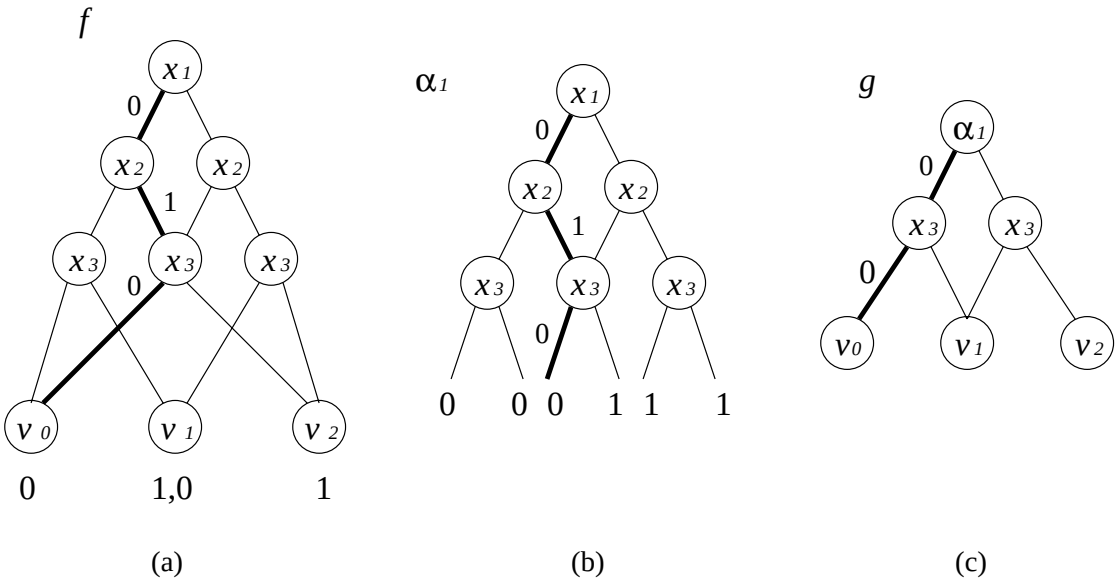


Figure 2.4: OBDD representing nondisjunctive decomposition

Chapter 3

Overlapping Variable Selection and Don't Care Assignment

In functional decomposition, initially, we have to select adequate bound set variables. In this problem, we apply the lambda set selection algorithm proposed in [7] for our variables partition. This method based on OBDD is to minimize the number of compatible classes. After we obtain bound set variables from this method, we have to select proper overlapping variables from the bound set to perform nondisjuntive decomposition. In previous researches [1, 6], overlapping variable is searched by putting candidates on the bottom of variable ordering of the bound set in OBDDs to compute the compatible classes of the OBDD. These method are very inefficient because for each candidate overlapping variable, OBDD needs to be re-constructed with respect to the new variable ordering. In this chapter, we will present an algorithm by only investigating the minterms of each compatible class. If implemented on an OBDD, our approach constructs OBDD just once. After the overlapping variable is selected in Section 3.1, we propose a don't care assignment algorithm in Section 3.2. These don't cares produced from overlapping can help to minimize the number of compatible classes in nondisjunctive decomposition.

3.1 Overlapping Variable Selection for Nondisjunctive Decomposition

3.1.1 Nondisjunctive Decomposition Chart

We first review the decomposition chart defined in [2] in this subsection.

Definition 3.1.1 A decomposition chart of function F is a 2-dimensional array. Its x-axis is indexed by variables in bound set while the y-axis is indexed by variables in free set. A column pattern in a decomposition chart forms a compatible class.

Example 3.1.1 illustrates a decomposition chart of a function.

Example 3.1.1 Let function $F(a, b, c, d, e, f)$ be partitioned to have bound set $\{a, b, c\}$ and free set $\{d, e, f\}$. Then, its decomposition chart and compatible classes are shown in Figure 3.1. In this decomposition chart, column and row are indexed by bound set $\{a, b, c\}$ and free set $\{d, e, f\}$, respectively. There are three column patterns and hence three compatible classes as denoted as X, Y, and Z.

For nondisjunctive decomposition, we will develop a modified decomposition chart called *nondisjunctive decomposition chart* which can characterizes the relations of overlapping variables and compatible classes.

Definition 3.1.2 A nondisjunctive decomposition chart of function F is a 2-dimensional array. Its x-axis is indexed by variables in bound set with the overlapping variable as the first variable of the index while the y-axis is indexed by variables in free set with the overlapping variable as the first variable of the index. Let the overlapping variable be v. Then, the upper-left corner submatrix gives the function value when $v = 0$ and the lower-right corner submatrix gives the function value when $v = 1$. The upper-right and lower-left submatrices are don't cares of the function because in these submatrices, index variable v are assigned $v = 0$ and $v = 1$.

F(a,b,c,d,e,f) *def* \ *abc*	000	001	010	011	100	101	110	111
000	0	1	0	0	0	0	1	0
001	0	1	0	0	0	0	1	0
010	0	0	1	1	0	0	0	1
011	0	1	0	0	0	0	1	0
100	1	1	1	1	1	1	1	1
101	1	0	1	1	1	1	0	1
110	1	1	0	0	1	1	1	0
111	1	0	1	1	1	1	0	1
	X	*Y*	*Z*	*Z*	*X*	*X*	*Y*	*Z*

compatible classes : *X*=(0,0,0,0,1,1,0,0)
Y=(1,1,0,1,1,0,1,0)
Z=(0,0,1,0,1,1,0,1)

Figure 3.1: The decomposition chart

Since the lower-left and upper-right submatrices are don't cares, we can assign any value to these submatrices. Hence, the number of compatible classes for $v = 0$ and $v = 1$ are determined by the upper-left and lower-right submatrices, respectively. A column pattern in upper-left and lower-right submatrices forms a compatible class for $v = 0$ and $v = 1$, respectively.

Take the function in Figure 3.2(a) as an example. In this example, function $F(a, b, c, d, e, f)$ is partitioned to bound set $\{a, b, c\}$ and free set $\{a, d, e, f\}$ with overlapping variable a. In this Figure, the upper-left submatrix represents the function when $a = 0$ and the lower-right submatrix represents the function when $a = 1$. There are three column patterns in the upper-left submatrix. Hence, there are three compatible classes when $a = 0$. Similarly, there are three compatible classes when $a = 1$.

Since overlapping variable will also appear in image function (the overlapping variable can be used to distinguish two compatible classes for $v = 0$ and $v = 1$ having the same code), we need only to ensure that the encoding of compatible classes (α functions) for $v = 0$ (or $v = 1$) is unique. That is, the same code can be assigned to the column pattern in $v = 0$ and $v = 1$. Therefore, the minimum number of α functions decomposed from F is logarithm of the maximum number of the compatible classes of $v = 0$ and $v = 1$ when v is selected as overlapping variable.

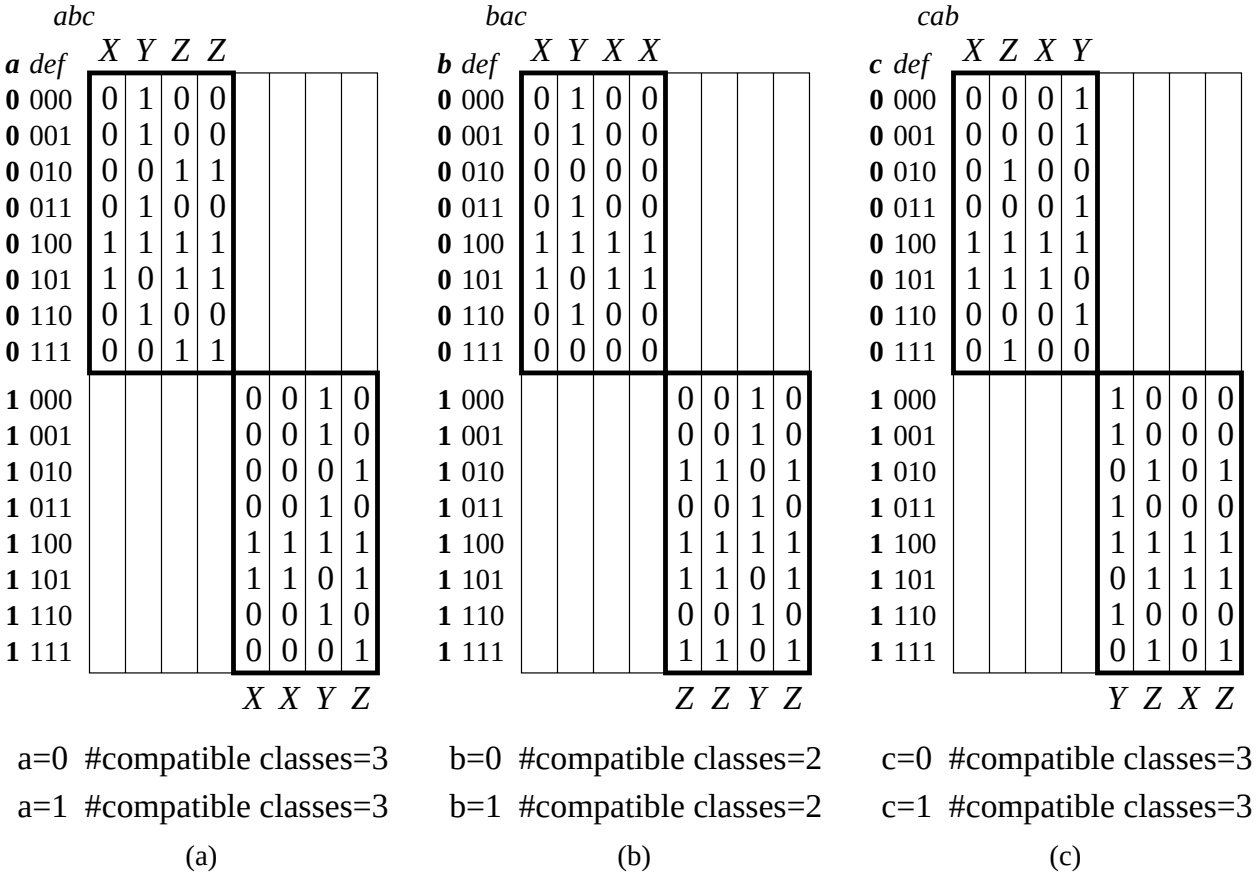


Figure 3.2: The nondisjunctive decomposition charts for different overlapping variables. (a) a is overlapping variable (b) b is overlapping variable (c) c is overlapping variable

Lemma 3.1.1 Given a function F and its overlapping variable v. The minimum of α functions decomposed from function F is logarithm of the maximum number of compatible classes of $v = 0$ and $v = 1$. ∎

For different overlapping variable, the resultant decomposition chart will be different. For example, in Figure 3.2, the nondisjunctive decomposition charts with a, b, or c is overlapped in the function are shown in Figure 3.2(a), (b) and (c), respectively. In this example, variable b is the best candidate to be selected as overlapping variable because the number of α functions can be reduced from 2 to 1 as compared to disjunctive decomposition.

3.1.2 Overlapping Variable Selection Algorithm

With respect to a disjunctive decomposition, we want to select one variable in bound set to be the overlapping variable. We do not want to construct different OBDD for different variable ordering corresponding to different overlapping variable as in [1]. We propose an algorithm that only examines the minterms in each compatible class to determine overlapping variable.

Let's examine the decomposition charts for disjunctive decomposition and nondisjunctive decomposition in Figure 3.1 and Figure 3.2(a). In Figure 3.2(a), variable a from bound set is selected as overlapping variable. Notice that in this figure all column patterns in upper-left submatrix ($a = 0$) and lower-right submatrix ($a = 1$) are column patterns from disjunctive decomposition chart. The appearance of each column pattern in submatrices is determined by the minterms in each compatible class. For example, there are three minterms $\{(0,0,0),(1,0,0),(1,0,1)\}$ in the compatible class corresponding to column pattern X in Figure 3.1. If a is selected as overlapping variable, minterm $\{(0,0,0)\}$ will appear in submatrix $a = 0$, and minterms $\{(1,0,0),(1,0,1)\}$ will appear in submatrix $a = 1$. Hence, column pattern X will

appear in submatrices for $a = 0$ and $a = 1$ Similarly, compatible classes corresponding to column patterns Y and Z both have minterms with $a = 0$ and minterms with $a = 1$. Hence, column pattern Y and Z will appear in both submatrices. Therefore, the column patterns for both submatrices $a = 0$ and $a = 1$ are $\{X, Y, Z\}$.

From the above observation, our overlapping variable selection algorithm proceeds as follows. First, with respect to a disjunctive input partitioning, we find the minterms in each compatible class. For each candidate overlapping variable, we check the value of minterms for the candidate variable in each compatible class. If the value is 0 (1), the column pattern corresponding to that compatible class will appear in the submatrix for the overlapping variable being 0 (1). Then, the overlapping variable is selected as the variable which results in the smallest maximum number of column patterns of two submatrices.

The result of running the overlapping variable selection algorithm on Example 3.1.1 is shown in Figure 3.3. In this Figure, we have three minterms, two minterms and three minterms in the compatible classes corresponding to column patterns X, Y, and Z, respectively. Three tables show the number of minterms appears in the upper-left and lower-right submatrices when a, b, and c are considered as overlapping variable. For example, in the second table, b is considered as overlapping variable. Since all minterms in compatible class corresponding to X column pattern have $b = 0$, we have three minterms for $b = 0$ and zero minterm for $b = 1$. Similarly, one minterm in compatible class corresponding to Y column pattern have $b = 0$ and one minterm have $b = 1$, we have one minterm for $b = 0$ submatrix and one minterm for $b = 1$ submatrix.

From these three tables, we can see that variable b will result in minimum number of α functions because there are only two column patterns for $b = 0$ and two column patterns for $b = 1$ and maximum of the two submatrices is two, which is smaller than

3 of the first table and 3 of the third table. The overall algorithm for overlapping variable selection is described in Figure 3.4.

minterms of each compatible class :

X= {(0,0,0),(1,0,0),(1,0,1)} Y= {(0,0,1),(1,1,0)} Z= {(0,1,0),(0,1,1),(1,1,1)}

#minterms	X	Y	Z	#compatible classes
$a = 0$	1	1	2	**3**
$a = 1$	2	1	1	**3**

#minterms	X	Y	Z	#compatible classes
$b = 0$	3	1	0	**2**
$b = 1$	0	1	3	**2**

#minterms	X	Y	Z	#compatible classes
$c = 0$	2	1	1	**3**
$c = 1$	1	1	2	**3**

Figure 3.3: Number of minterms with respect to different overlapping variable

3.2 Compatible Class Pairing

As shown in Figure 3.2, upper-right and lower-left submatrices are don't care. However, before we synthesize image functions, don't care needs to be assigned values. There are two objectives of the don't care assignment. The first objective is that the number of column patterns after don't care assignment can not exceed the maximum of column patterns of upper-left and lower-right submatrices. The second objective is to increase the decomposability of image functions. To these ends, we develop a bi-partite matching algorithm.

Since after don't care assignment, the number of column patterns of the whole matrix can not exceed the maximum number of column patterns of upper-left and

Input : Bound Set B and Compatible Class Functions $fc's$;
Output: Overlapping Variable v ;
Internal: $N_{i=0}, N_{i=1}, max_i$;
Begin
(1) **For** each variable $i \in$ bound set B ;
(2) $N_{i=0}$ is the number of compatible classes with $i = 0$;
(3) $N_{i=1}$ is the number of compatible classes with $i = 1$;

(4) **For** each variable $i \in$ bound set B ;
(5) max_i is the large number between $N_{i=0}$ and $N_{i=1}$;
/* max_i is the number of compatible classes for i is overlapped*/

(6) **For** all the max_i ;
(7) The minimal value of max_i is max_v;
(8) **return** v ;
End

Figure 3.4: An Overlapping Variable Selection Algorithm

lower-right submatrices, for don't care in lower-left submatrix, we can select only column patterns from lower-right submatrix. Similarly, for don't care in upper-right submatrix, we can select only column patterns from upper-left submatrix. Moreover, once a column in lower-left submatrix whose corresponding column in the upper-left have a column pattern P_1 select a column pattern P_2 from lower-right, then the upper-right column whose corresponding column of lower-right column have column pattern P_2 must select P_1 as its column pattern. We take the nondisjunctive decomposition chart with the best overlapping result in Figure 3.2(b) as an example. If the first column of the submatrix in lower-left whose corresponding column in upper-left has column pattern X is assigned column pattern Z, then columns in upper-right whose corresponding columns in lower-right have column pattern Z must be assigned X as in Figure 3.5. From the above description, the don't care assignment problem is a bi-

partite matching problem. We model the problem as follows. We build a bi-partite graph $G(V, U, E)$ where the column patterns (compatible classes) in the upper-left submatrix corresponds to a vertex v_i in V, the column pattern (compatible classes) in the lower-right submatrix corresponds to a vertex u_i in U, and each pair of vertices of (v_i, u_i) is connected by an edge. For instance, the example in Figure 3.2(b) has a bi-partite graph as shown in Figure 3.6. Nodes X and Y in V correspond to the column patterns in the upper-left submatrix of Figure 3.2(b), and nodes Y and Z in U correspond to the column patterns in the lower-right submatrix of Figure 3.2(b).0 There are 4 connections of (X, Y), (X, Z), (Y, Y), and (Y, Z) between V and U in the graph. Each connection represents a paired compatible class.

bac	X	Y	X	X				
b def								
0 000	0	1	0	0				
0 001	0	1	0	0				
0 010	0	0	0	0				
0 011	0	1	0	0	X	X		X
0 100	1	1	1	1				
0 101	1	0	1	1				
0 110	0	1	0	0				
0 111	0	0	0	0				
1 000					0	0	1	0
1 001					0	0	1	0
1 010					1	1	0	1
1 011	Z		Z	Z	0	0	1	0
1 100					1	1	1	1
1 101					1	1	0	1
1 110					0	0	1	0
1 111					1	1	0	1
					Z	Z	Y	Z

Figure 3.5: Compatible class pairing in nondisjunctive decomposition chart

Now, we have model our problem as a bi-partite matching problem. A matching will correspond to a don't care assignment which achieve our first objective. However,

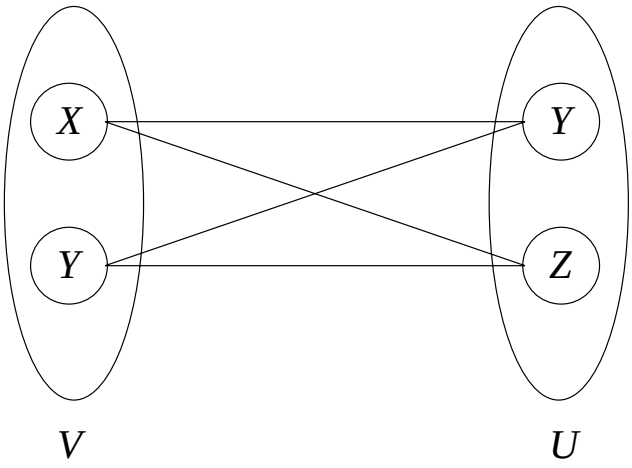


Figure 3.6: A bi-partite graph in Example 3.1.1

for the same graph, there may be many matchings and different matching may result in different decomposability of the next level image function. Hence, to achieve the second objective, we will assign weight on each edge of a bi-partite graph. The weight is assigned to reflect the decomposability of the next level image function. Our weight assignment heuristic is developed inspired by the compatible classes encoding method in [2]. Before we present our heuristic, we first give the following definition.

Definition 3.2.1 A *partition vector* $\pi_\chi(F)$ of function F with bound set χ is denoted as $< s_0, ..., s_k >$ for $k = 2^{|Bound\_Set|} - 1$ where $< s_0, ..., s_k >$ is a symbolic notation of n column patterns $< c_0, ..., c_{n-1} >$ and element s_i equals to s_j if and only if the column pattern c_i equals to column pattern c_j.

For example, in Example 3.1.1, with respect to bound set $\chi = \{a, b, c\}$, the partition vector $\pi_\chi(F)$ of F is $< 0, 1, 2, 2, 0, 0, 1, 2 >$. Furthermore, as for the compatible function $X(d, e, f)$ of F, its partition vector $\pi_\chi(X)$ with bound set $\chi = \{d, e\}$ is $< 0, 0, 1, 0 >$, which is shown in Figure 3.7.

Now, we explain our edge weight assignment heuristic algorithm. First, we must decide which variables to be in the bound set when the image function in the next

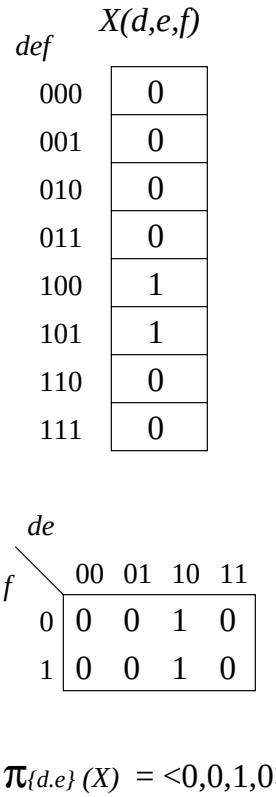


Figure 3.7: Partition of compatible function X with bound set $\chi = \{d, e\}$

level is decomposed. From [2], we know that only variables in the free set of the current decomposition need to be considered. The bound set of image function is selected from free set of the current partition using λ set selection algorithm in [7].

Now, for a given bound set χ, we want to know the decomposability of the whole column pattern in which one half of the column having column pattern P_1 is in care set and the other half of the column is don't care and assigned certain pattern P_2. The decomposability can be determined by the number of column patterns with respect to the bound set χ after decomposing the paired columns. The number of patterns of decomposing the combined column of P_1 and P_2 is computed by concatenating the *partition vectors* of P_1 and P_2 and compute the number of patterns.

For example, in Figure 3.5, if column X is paired with Z, we will have a column pattern denoted as (X, Z) as shown in Figure 3.8. If bound set is selected as $\chi = \{d, e\}$, then the number of column patterns of the column (X, Z) with this bound set is 4 as shown in Figure 3.8. The number is computed by concatenating the *partition vectors* of X and Z.

Therefore, to compute the weight on an edge is to compute *partition vectors* for columns in upper-left and lower-right submatrices with respect to the selected bound set χ. And, the decomposability of pairing two columns is to compute the number of column patterns of the concatenated *partition vectors* of the two columns.

We take Example 3.1.1 to illustrate the computation of the weight on each edge. Suppose we select $\{d, e\}$ as bound set for the subsequent decomposition. There are totally three column patterns X, Y and Z in upper-left and lower-right submatrices. With respect to the bound set $\chi = \{d, e\}$, the three columns are decomposed as shown in Figure 3.9.

If the column X is paired with Y, the decomposition chart of (X, Y) is shown in Figure 3.10(a) and the number of column patterns with bound set $\chi = \{d, e\}$ is 4. Similarly, paring (Y, Y) (X, Z), and (Y, Z) are shown in Figure 3.10(b)(c)(d) and the number of column patterns for these three pairings are 3, 4, and 4, respectively. Therefore the weight assigned on each edge of the bi-partite graph is shown in Figure 3.10(e). The best matching is found by minimal cardinality matching of the bi-partite graph G. If the number of vertices in the two parties of the bi-partite graph is not the same, then there will be vertices without mapping edge. In this case, connect them to the vertices in the opposite party with minimum cost on the edge.

f \ de	X: 00	01	10	11
0	0	0	1	0
1	0	0	1	0

$\pi_{\{d,e\}}(X) = \langle 0,0,1,0 \rangle$

f \ de	Z: 00	01	10	11
0	0	1	1	0
1	0	0	1	1

$\pi_{\{d,e\}}(Z) = \langle 0,3,1,2 \rangle$

b def	
0 000	0
0 001	0
0 010	0
0 011	0
0 100	1
0 101	1
0 110	0
0 111	0
1 000	0
1 001	0
1 010	1
1 011	0
1 100	1
1 101	1
1 110	0
1 111	1

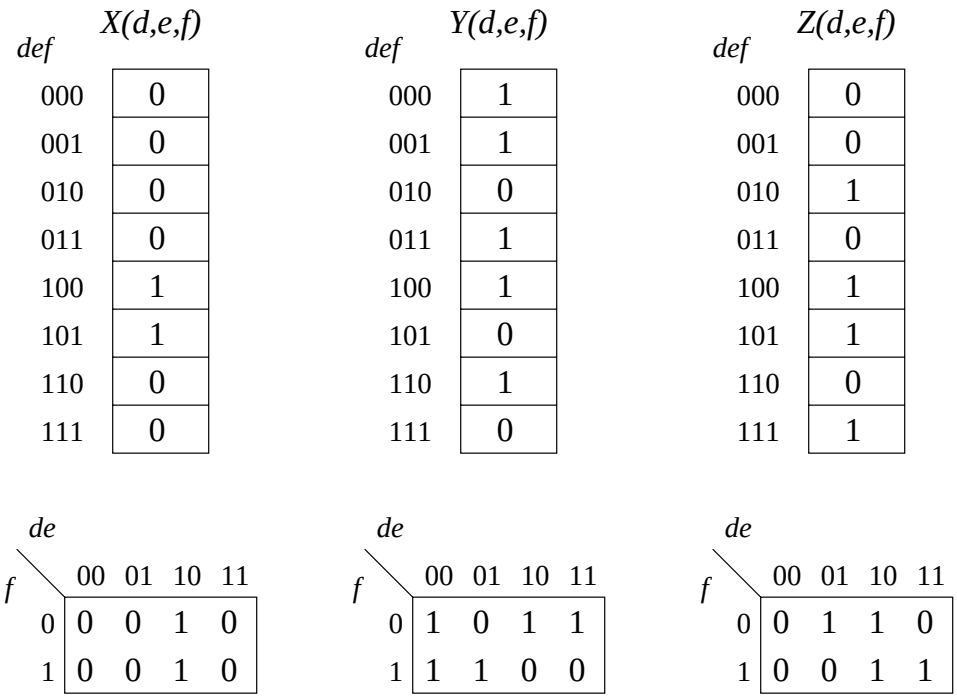
X(d,e,f)

Z(d,e,f)

f \ bde	X				Z			
	0	0	1	0	0	1	1	0
	0	0	1	0	0	0	1	1

$\pi_{\{d,e\}}(XZ) = \langle 0,0,1,0,0,3,1,2 \rangle$

Figure 3.8: Column patterns of the paired compatible class (X, Z)



(a) $\pi_{\{d.e\}}(X)$ =<0,0,1,0> (b) $\pi_{\{d.e\}}(Y)$ =<1,2,3,3> (c) $\pi_{\{d.e\}}(Z)$ =<0,3,1,2>

Figure 3.9: The partition chart of each compatible class

f \ de	X				Y			
	0	0	1	0	1	0	1	1
	0	0	1	0	1	1	0	0

$\pi_{\{d,e\}}(XY)$ = <0,0,1,0,1,2,3,3>

(a)

f \ de	Y				Y			
	1	0	1	1	1	0	1	1
	1	1	0	0	1	1	0	0

$\pi_{\{d,e\}}(YY)$ = <0,1,2,2,0,1,2,2>

(b)

f \ de	X				Z			
	0	0	1	0	0	1	1	0
	0	0	1	0	0	0	1	1

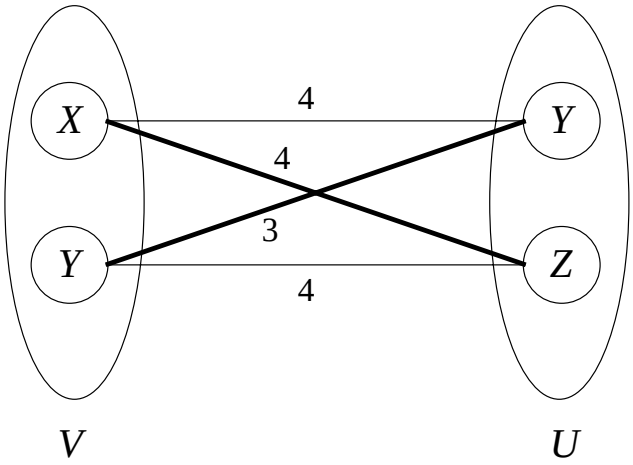
$\pi_{\{d,e\}}(XZ)$ = <0,0,1,0,0,3,1,2>

(c)

f \ de	Y				Z			
	1	0	1	1	0	1	1	0
	1	1	0	0	0	0	1	1

$\pi_{\{d,e\}}(YZ)$ = <1,2,3,3,0,3,1,2>

(d)



(e)

Figure 3.10: Minimal cardinality matching

Chapter 4

Compatible Class Encoding

In this chapter, we will discuss the encoding problem of nondisjunctive decomposition. Instead of reducing the number of cubes or literals in the image function as suggested in [1, 8], our heuristic will put emphasis on reducing the number of compatible class of the decomposition at the next level suggested in the HYDE algorithm [2].

4.1 Encoding in Functional Decomposition

After we finish our compatible class pairing procedure, the next step is to perform encoding of the paired compatible classes. First, an encoding chart is introduced which is used to show the relation between encoding and the number of compatible classes of the decomposition at the next level image function.

Definition 4.1.1 An encoding chart is a 2-dimensional array used to encode functions. It is similar to the decomposition chart except that it is indexed by the encoding bits. Its x-axis is indexed by bits that are selected as the bound set of the next level decomposition and its y-axis is indexed by bits that are selected as the free set of the next level decomposition.

In [2], it has been shown that the encoding of α functions will have no effect on the decomposability of the image function if all α bits are partitioned into the same

set. Hence, the following theorem was derived.

Theorem 4.1.1 The α bits must be separated into bound set and free set of the decomposition at the next level image function such that the encoding can have effects on the number of compatible classes in the subsequent decomposition. ∎

Figure 4.1 illustrate an example of our encoding problem. Let function F be partitioned to have three compatible classes X, Y, and Z. Therefore, two α-bits, α_0 and α_1, are needed to encode these three compatible classes and the image function is a five-input function $g(\alpha_0, \alpha_1, d, e, f)$. Let the next level image function be partitioned so that $\{d, e\}$ is in the bound set. Suppose we choose α_0, d, and e as the bound set and α_1 and f as the free set of the decomposition for function $g(\alpha_0, \alpha_1, d, e, f)$. Then, without considering the encoding bits of α functions, the partitioned X, Y, and Z with respect this partitioning is shown in Figure 4.1(a). Assume that X, Y, and Z are encoded as $(0, 0)$, $(1, 0)$, and $(0, 1)$ where the two bits corresponds to α_0 and α_1 functions, respectively. Then, its encoding chart is shown in the left of Figure 4.1(b) and the corresponding decomposition chart is shown in the right, where the symbol " $*$ " means don't care. In this case, we will have six column patterns of the decomposition of the g-function. However, if we choose the codings (α_0, α_1) as $(1, 1)$, $(0, 0)$, $(1, 0)$ for X, Y, Z as shown in Figure 4.1(c), we will have only four column patterns in the decomposition of the g-function.

From the above example, we know that the encoding will affect the number of compatible classes of the decomposition for the g-function. The fewer the number of compatible classes is produced in the decomposition of the image function, the fewer bits are needed to encode these compatible classes of the next decomposition. Moreover, it can be seen that encoding problem is in effect the determination of the position of X, Y and Z in the encoding chart. The following theorem was derived.

Theorem 4.1.2 Once variables in the bound set of the decomposition for an image function have been selected, determining which compatible class functions in the same column and in the same row in the encoding chart will decide the number of compatible classes of the image function. The exact codes of these columns and rows do not affect the number of compatible classes of the image function. ∎

In the following, we review how HYDE [2] performs the encoding. In its column-set and row-set combination approach, they evaluate which compatible classes should be bound in the same column or in the same row to the encoding chart. Compatible classes with the same positions having same column patterns are preferred to be bound to the same column. By this way they get smaller set of column patterns in the encoding chart. However, involving the row-set combination approach may break the well-mapped results of column sets. For Figure 4.2, There are ten compatible classes with their partitions $\pi_0, ..., \pi_9$. After their column-set and row-set combination procedures, the column sets are $\{\pi_3, \pi_4, \pi_6, \pi_8\}$, $\{\pi_2, \pi_7\}$, $\{\pi_0\}$, $\{\pi_1\}$, $\{\pi_5\}$, and $\{\pi_9\}$, and the row sets are $\{\pi_7, \pi_8\}$, $\{\pi_5, \pi_6\}$, $\{\pi_2, \pi_4\}$, $\{\pi_0, \pi_9\}$, and $\{\pi_1, \pi_3\}$. In its encoding chart, the partitions π_1 and π_5 were inserted into the column between π_2 and π_7, which increase the number of column patterns of the column set $\{\pi_2, \pi_7\}$ from 3 to 4.

For this reason, We raise an column pattern matching algorithm that can fairly reflect on our heuristic without corrupting well-mapped column-set combinations.

4.2 Column Pattern Matching Algorithm

In order to reduce the number of compatible classes in the encoding chart, we propose a method to form new column vectors based on the column patterns previously formed.

Initially, we apply a heuristic similar to column-set combination cited above to select one column set which has minimal number of column patterns in the decompo-

sition chart. We call it *primary column pattern set*. Then we form the next column vector based on the column patterns in the primary column pattern set. We repeat this step until all the compatible classes are placed into the encoding chart. In this way, the previously well-formed column sets will not be destroyed by the later step. Moreover, the don't care blocks can be utilized more efficiently. We described these steps of our pattern-matching algorithm in more detail in the following section and take Example 4.2.1 for demonstration.

Example 4.2.1 Assume we have ten compatible class functions, $f_{P_0}, ..., f_{P_9}$, with their partitions $\pi_0, ..., \pi_9$ respectively as follows.

$\pi_0 =< 0,3,3,0 >, \pi_1 =< 1,2,2,3 >, \pi_2 =< 3,0,0,1 >, \pi_3 =< 0,1,0,2 >,$

$\pi_4 =< 1,2,1,3 >, \pi_5 =< 2,3,2,1 >, \pi_6 =< 2,0,1,2 >, \pi_7 =< 3,2,0,3 >,$

$\pi_8 =< 0,0,3,1 >, \pi_9 =< 1,2,3,0 > .$

Step 1 Taking the maximal column set with minimal column patterns to be the primary column set and indicating the produced primary column patterns.

The same concept as in column-set combination in HYDE algorithm is invoked. Binding the compatible classes with the same positions having identical partition elements into a column, will reduce their column patterns to minimal. In order to find the maximal set of compatible classes with the same positions having same partition elements, we denote position i in a partition as η_i for convenience. In Example 4.2.1, for partition π_0, the contents of η_1 and η_4 are the same, and the contents of η_2 and η_3 are the same, we say that positions with the same content of π_0 are $\eta_1\eta_4$ and $\eta_2\eta_3$. π_1 also have same contents in $\eta_2\eta_3$. So we can put π_0 and π_1 in the same column to reduce number of column patterns. We record these position information in Figure 4.3. The gain of each set is *(number of positions with same content)* $\times$ *(number of partitions in the set)*. Find the maximal gain column set.

Table 4.1: The primary column pattern set

$\prod_0$	0 3 3 0
$\prod_1$	1 2 2 3
$\prod_2$	3 0 0 1

Table 4.2: The remaining compatible classes

$\prod_3$	0 1 0 2
$\prod_4$	1 2 1 3
$\prod_5$	2 3 2 1
$\prod_6$	2 0 1 2
$\prod_7$	3 2 0 3
$\prod_8$	0 0 3 1
$\prod_9$	1 2 3 0

In Example 4.2.1, the column sets with minimal number of column patterns are $\{\pi_0, \pi_6, \pi_7\}$ $\{\pi_0, \pi_1, \pi_2\}$ and $\{\pi_3, \pi_4, \pi_5\}$. Suppose we select $\{\pi_0, \pi_1, \pi_2\}$ to be the primary column set, and put π_0, π_1, π_2 from top to bottom in the first column of the encoding chart. Then, we have the primary column pattern set shown in Table 4.1 and the remaining π's to be placed are shown in Table 4.2. The primary column patterns in the encoding chart is also shown in Figure 4.4(a) and the three column patterns are X, Y, and Z.

Step 2 Compute the weight of placing a π on a particular row of the next column in the encoding chart. The weight is defined to reflect the similarity of the present π and the primary column patterns at the corresponding row position. The weight is computed as the number of elements in the π that is the same as the elements of the primary column pattern in the same row. In the new column, the partition π with the maximal weight is selected as candidate to be put on that row.

In Figure 4.4(a), assume that we are to decide which π to be put in the second

column of the encoding chart. For example, if π_7 is put in the first row, since $\pi_7 =< 3, 2, 0, 3 >$ has 2 elements, "3" and 1 element of "0" that are the same as the elements in $\pi_0 =< 0, 3, 3, 0 >$, its weight is 3. Similarly, we can compute the weight for putting other π in the first row and the results are shown in Figure 4.4(b). Since π_7 and π_8 have the maximal weight, they are selected as candidates to be placed on row 1.

Similarly, we can compute candidates to be put in row 2 and row 3. Since π_4 and π_5 have the maximal weight of 4, they are selected as candidates to be put in row 2. Finally, π_8 is selected to be put in row 3.

Step 3 For each column pattern in the primary column pattern set, find out $useful$ π combinations from all the combinations of the candidate sets.

A $useful$ π combination means that a column in the encoding chart formed from the candidate set will have the same column patterns as the patterns in the primary column pattern set. From Step 2, we have two candidates for row 1, two candidates for row 2, and one for row 3 as shown in the encoding chart of Figure 4.5(a). There are $3 \times 3 \times 2 - 1 = 17$ combinations to form the column patterns for the new column of the encoding chart. However, only a part of them are useful. For example, if π_8 and π_4 are selected from candidate set of row 1 and row 2, we will have a new column, $(\pi_8, \pi_4, *, *)$, where * means don't care in the encoding chart (that is four columns in the decomposition chart) as shown in Figure 4.5(b). For the first column of the four newly formed columns, this combination is useful because the first column pattern is the same as a pattern X in the primary pattern set. Similarly, combination of π_7, π_5, π_8 from row 1, row 2 and row 3, and combination of π_8 from row 1 are useful for the first column because the first column of the first combination is the column pattern Y and the first column of the second combination is the column pattern Z as shown in Figure 4.5(c) and (d). However, also in Figure 4.5(e), combination such as $(\pi_8, \pi_5, *, *)$ is not useful for the first column because its first column pattern is

$< 0, 2, *, * >$ which is not the same as any pattern in the primary column pattern set. Note that if a useful column combination has only one element in it such as the combination of $(\pi_8, *, *, *)$, we will ignore this combination because too much don't care will be used.

For the first, second, third and fourth newly formed column, we compute their useful combinations. For example, for the first column, we have two combinations of $(\pi_8, \pi_4, *, *)$ and $(\pi_7, \pi_5, \pi_8, *)$. For the second column, we have two combinations of $(*, \pi_4, \pi_8, *)$ and $(\pi_8, \pi_5, *, *)$. For the third column, we have two combinations of $(\pi_7, \pi_4, \pi_8, *)$ and $(\pi_8, \pi_5, *, *)$. For the fourth column, we have one combination of $(*, \pi_4, \pi_8, *)$. The results are shown in Figure 4.6. Among them, there are only 5 different combinations $(*, \pi_4, \pi_8, *)$, $(\pi_8, \pi_4, *, *)$, $(\pi_8, \pi_5, *, *)$, $(\pi_7, \pi_4, \pi_8, *)$, and $(\pi_7, \pi_5, \pi_8, *)$.

Step 4 Select the best combination. The gain of each combination is computed as *(the number of elements in the combination)*$\times$ *(the number of columns this combination appears)* . The combination with the maximal gain is selected as the next column vector in the encoding chart because it will produce maximal number of primary column patterns.

For instance, there are 2 elements in $(*, \pi_4, \pi_8, *)$, and this combination appears in three columns (the secord, the third, and the fourth columns) in Figure 4.6, twice for $(*, \pi_4, \pi_8, *)$, and once for the subvector of $(\pi_7, \pi_4, \pi_8, *)$. Therefore, the gain of this combination is $2 \times 3 = 6$. We compute the gains of all other column combinations,

$$gain(\pi_8, \pi_4, *, *) = 2 \times 1 = 2$$

$$gain(\pi_8, \pi_5, *, *) = 2 \times 2 = 4$$

$$gain(\pi_7, \pi_4, \pi_8, *) = 3 \times 1 = 3$$

$$gain(\pi_7, \pi_5, \pi_8, *) = 3 \times 1 = 3$$

Since the column vector $(*, \pi_4, \pi_8, *)$ has the maximal gain, it is selected as the next column vector in the encoding chart. In this newly formed column, there are 4 column patterns $< *, 1, 0, * >$, $< *, 2, 0, * >$, $< *, 1, 3, * >$, and $< *, 3, 1, * >$. Only the pattern $< *, 1, 0, * >$ is not in the primary pattern set and is a new column pattern.

Step 5 Update the set of primary column patterns by adding the newly created patterns. If there still have free columns in the encoding chart and unassigned π, go to Step 2. Otherwise, if there are no free columns for assignment, the remaining partitions are assigned to free blocks in the column according to the initial column-set combination result and in the row with the most number of don't care blocks. This procedure is terminated when all π are assigned.

For the same example, in the next iteration, $(*, \pi_5, \pi_3, *)$ is selected as the third column of the encoding chart and in the last iteration $(*, \pi_9, \pi_6, *)$ is selected as the last column. There is still one unassigned π (π_7) which is assigned to the column as the partitions (π_6) together in the same column set in step 1, and to the row with maximal free blocks in it (the last row).

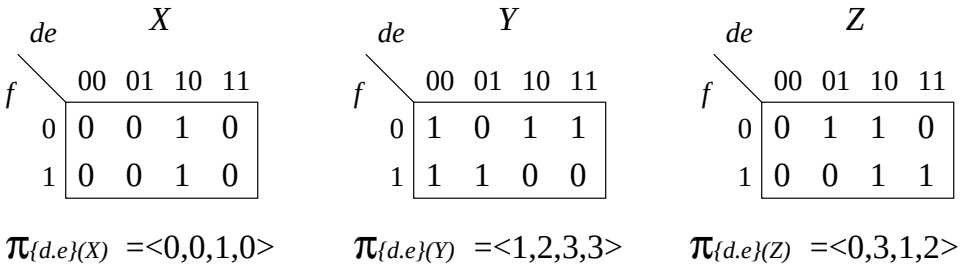
Finally, we have our encoding chart as shown in Figure 4.7. In this chart, we have only six column patterns which is smaller than the result of HYDE in Figure 4.8, which has 8 column patterns for the next decomposition.

4.3 Application to LUT based FPGA Synthesis

FPGA is an important technology in VLSI designs because of its short design cycle and low manufacturing cost. Among different FPGA's, LUT-based FPGA is one of the most popular architecture which has been used in many commercial products such as Xilinx [19] 3000/4000 series. In an LUT-based FPGA, the basic programmable

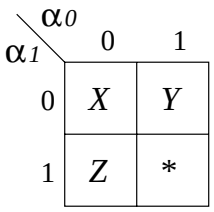
logic block is a k-input lookup table which can implement any Boolean function of up to k variables. The previous LUT-based FPGA mapping algorithms can be divided into three classes. The first class emphasizes on minimization the number of LUT's in the mapped results [2, 6, 8, 9]. The second class stresses on the delay minimization of the mapped solutions [10, 11, 12, 13]. The third class emphasizes on maximization of routability [17, 18].

Functional decomposition is one of most popular synthesis method to extract k-input subfunction for look-up table based FPGA. We will apply our nondisjunctive decomposition to synthesize LUT-based FPGA. Our nondisjunctive decomposition will transform a Boolean network into a functionally equivalent network of k-input LUT's under area consideration. First, we choose the bound set size as the input size k of a LUT and pick the overlapping variable from the bound set that will reduce the number of α-functions. Then we apply our encoding algorithm to each α-function to get a g-function with a minimal number of compatible classes for next decomposition.



$\pi_{\{d.e\}(X)}$ =<0,0,1,0> $\pi_{\{d.e\}(Y)}$ =<1,2,3,3> $\pi_{\{d.e\}(Z)}$ =<0,3,1,2>

(a)

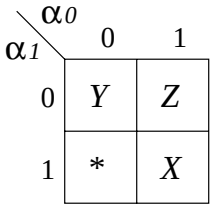


The encoding chart

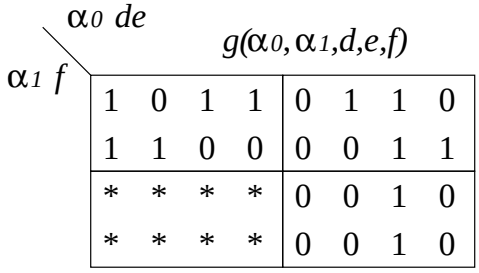
The decomposition chart

(at least 6 compatible classes)

(b)



The encoding chart



The decomposition chart

(at least 4 compatible classes)

(c)

Figure 4.1: Relations between compatible classes and encoding: (a) three compatible classes X, Y, and Z with their partitions of $\{d, e\}$ as bound set (b) the encoding chart and the decomposition chart for one case (c) the encoding chart and the decomposition chart for the other case

column sets : { π_3 π_4 π_6 π_8 } { π_2 π_7 } { π_0 } { π_1 } { π_5 } { π_9 }

row sets : { π_7 π_8 } { π_5 π_6 } { π_2 π_4 } { π_0 π_9 } { π_1 π_3 }

The encoding chart

π_7 <1 1 2 1>	π_8 <1 2 1 2>		
π_5 <0 1 0 2>	π_6 <1 0 0 0>		
π_2 <3 0 1 3>	π_4 <0 1 3 1>		
π_1 <0 2 1 3>	π_3 <2 1 0 1>	π_0 <0 1 2 3>	π_9 <3 2 1 0>

Figure 4.2: Graph formulation of column-set combination

positions with the same content	partitions	gain
$\eta_1\eta_4$	π_0 π_6 π_7	6
$\eta_2\eta_3$	π_0 π_1 π_2	6
$\eta_1\eta_3$	π_3 π_4 π_5	6
$\eta_1\eta_2$	π_8	2

Figure 4.3: Partitions and positions with the same contents

row1	π_0 <0 3 3 0>			
row2	π_1 <1 2 2 3>			
row3	π_2 <3 0 0 1>			
row4				

<X Y Y Z>

(a)

remaining partitions list	weight for putting on row 1 adjacent to π_0 <0 3 3 0>
π_3 <0 1 0 2>	2
π_4 <1 2 1 3>	1
π_5 <2 3 2 1>	1
π_6 <2 0 1 2>	1
π_7 <3 2 0 3>	**3**
π_8 <0 0 3 1>	**3**
π_9 <1 2 3 0>	2

(b)

Figure 4.4: The encoding chart and weight information to each partition: (a) primary column set placed in the first column of the encoding chart (b) is a weight-table for remaining partitions to be placed in row 1

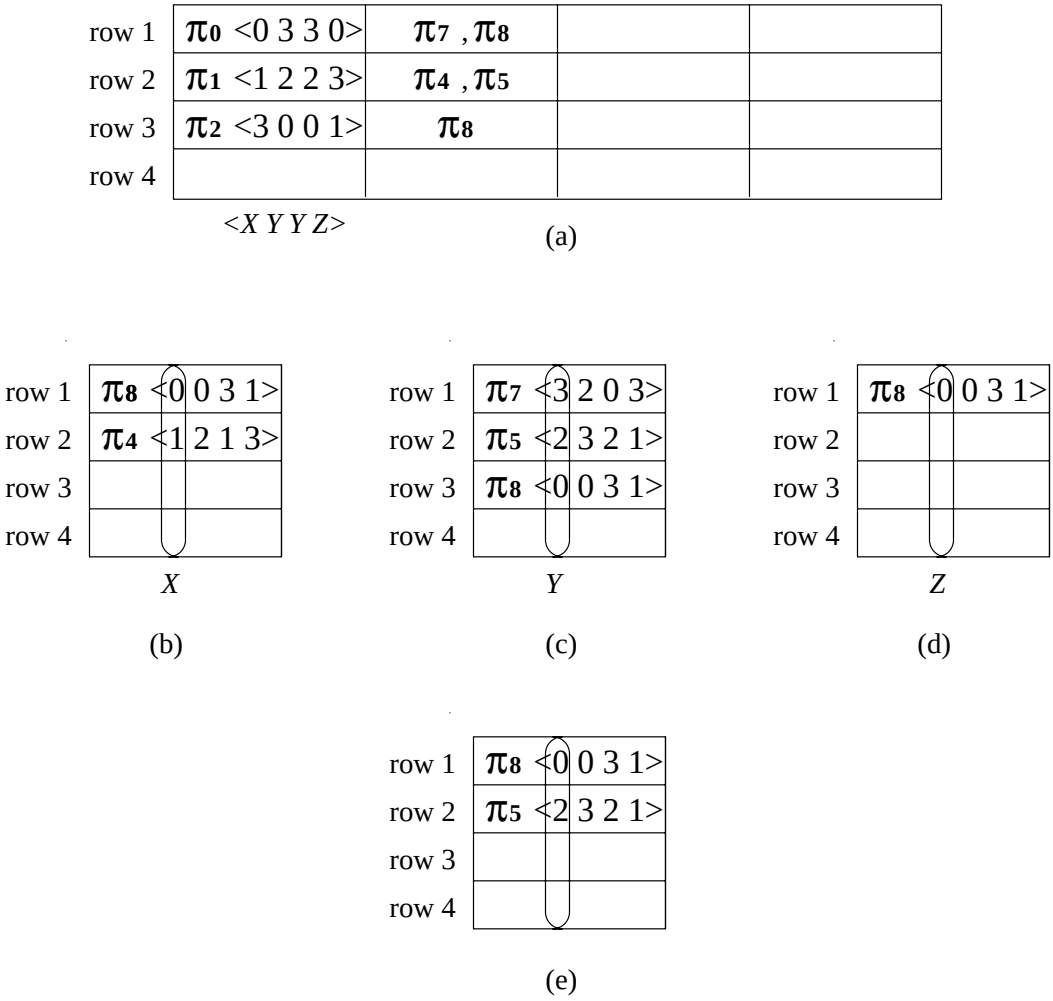


Figure 4.5: The column vectors forming the primary column patterns in the first column

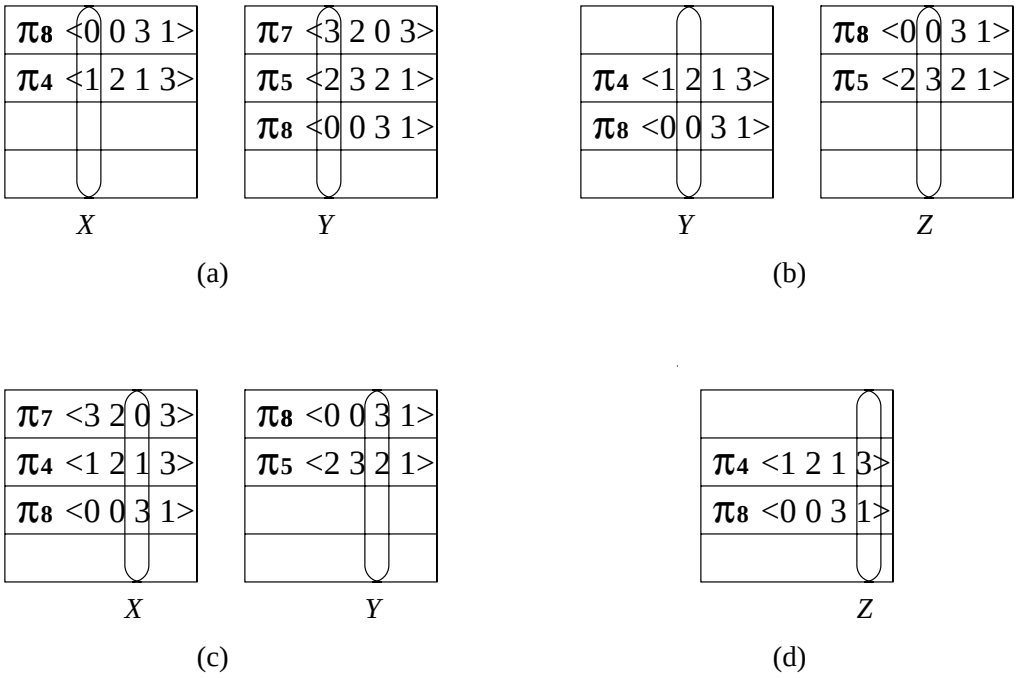


Figure 4.6: Column vectors for each column: (a), (b), (c) and (d) are useful partition combinations to form the primary column patterns X, Y, and Z in column 1, 2, 3 and 4

The encoding chart

π_0 <0 3 3 0>			
π_1 <1 2 2 3>	π_4 <1 2 1 3>	π_5 <2 3 2 1>	π_9 <1 2 3 0>
π_2 <3 0 0 1>	π_8 <0 0 3 1>	π_3 <0 1 0 2>	π_6 <2 0 1 2>
			π_7 <3 2 0 3>
<0 1 1 2>	<3 1 0 2>	<1 2 1 4>	<4 1 2 5>

Figure 4.7: Final result of the encoding chart

The encoding chart

π_0 <0 3 3 0>	π_4 <1 2 1 3>	π_7 <3 2 0 3>	π_9 <1 2 3 0>
π_1 <1 2 2 3>	π_5 <2 3 2 1>		
π_2 <3 0 0 1>	π_3 <0 1 0 2>	π_6 <2 0 1 2>	π_8 <0 0 3 1>

<0 1 1 2> <3 4 3 5> <5 6 2 5> <3 6 7 2>

Figure 4.8: The result of Encoding in HYDE algorithm

Chapter 5

Experimental Results

Our overlapping decomposition algorithm has been implemented for LUT based FPGA's synthesis under SUN-Ultra 60 workstation. The implemented software is embedded in SIS environment. A set of experiments are carried out on MCNC benchmark circuits. Initial circuits are first optimized by SIS to obtained a minimized technology independent circuit descriptions. For small circuits, they are collapsed to two-level representations. For large circuits, they are optimized by SIS algebraic script. Then, the optimized initial circuits are technology-mapped to five-input one-output RAM-based FPGA using the following script. For two-level circuits, the script is: *our decomposition, xl_partition -tm, xl_cover* and for multi-level circuits: *simplify, our decomposition, xl_partition, xl_cover*. For multi-level circuits, the script are applied several times to improve the results by taking advantage of extracting the local don't care set.

We compare our results to those produced by other synthesis techniques reported in [1] and [2], where [1] used not only disjunctive decomposition but nondisjunctive decomposition and encoded compatible classes in a straightforward way: assigning the binary representation of i to the i-th element. We only list its decomposition results without resubstitution. Column 3 are the results of HYDE algorithm [2] that only used disjunctive decomposition. These results are shown in Table 5.1.

Table 5.1: Results for 5-input 1-output LUT's

Circuit			**no resub. in [1]**	**HYDE**	**My Decomp**
name	in	out	LUT	LUT	LUT
5xp1	7	10	15	13	13
9sym	9	1	7	6	7
alu2	10	6	48	50	54
alu4	14	8	172	206	199
apex6	135	99	192	186	173
apex7	49	37	120	54	50
b9	41	21	53	36	30
clip	9	5	18	14	15
duke2	22	29	175	116	103
f51m	8	8	12	12	15
misex1	8	7	12	13	13
misex2	25	18	40	29	28
rd73	7	3	8	6	7
rd84	8	4	12	9	10
sao2	10	4	23	22	21
vg2	25	8	44	18	17
z4ml	7	4	6	5	5
Total			957	795	760

From this table, it can be seen that on an average, our algorithm produces about 21% less number of logic cells as compared to [1] and about 5% less number of logic cells as compared to HYDE. It can be noted that our overlapping decomposition is not suitable for all circuits. It is especially good for arithmetic type of circuits with some control signals. We can see that our overlapping advantage is more apparent in the large circuits.

Chapter 6

Conclusions

Nondisjunctive decomposition is a functional decomposition method for logic synthesis. In some cases, adopting not only disjunctive decomposition but nondisjunctive decomposition will have better results if it can derive less image functions than pure disjunctive decomposition. The way to selecting a beneficial overlapping variable for nondisjunctive decomposition is first proposed in this thesis. We also solve the compatible class encoding problem for our nondisjunctive decomposition approach. Application of this method to the synthesis of LUT-based FPGA's was discussed. Experimental results show that our approach is practical and promising in particular benchmarks.

Bibliography

[1] Hiroshi Sawada, Takayuki Suyama , and Akira Nagoya, "Logic Synthesis for Look-Up Table based FPGAs using Functional Decomposition and Support Minimization," in *Proc. Int. Conf. on CAD*, pp. 353-358, Nov. 1995.

[2] Jie-Hong R. Jiang, Jing-Yang Jou, Juinn-Dar Huang, "Compatible Class Encoding in Hyper-Function Decomposition for FPGA Synthesis," *Proc. of DAC'98*, pp. 712-717, June. 1998.

[3] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. Int. Symp Circuits Syst.,* , pp. 49-54, May. 1982.

[4] J. P. Roth and R, M, Karp, "Minimization over Boolean graphs," *IBM Journal*, pp. 227-238, 1962.

[5] S-C. Chang and M. Marek-Sadowska, "Techmology mapping via transformations of function graph," in *Proc. Int. Conf. Computer Design*, pp. 159-162, Oct. 1992.

[6] Yung-Te Lai, Kuo-Rueih Picky Pan, and Massoud Pedram, "OBDD-based functional decomposition algorithms and implementation," in *IEEE Trans. CAD*, vol. 15, pp. 977-990, August. 1996.

[7] Jie-Hong R. Jiang, Jing-Yang Jou, Juinn-Dar Huang and Jung-Shian Wei, "BDD Based Lambda Set Selection in Roth-Karp Decompsition for LUT Architecture," *Proc. ASP-DAC*, pp. 259-264, January. 1997.

[8] Rajeev Murgai, Robert K Brayton, and Alberto Sangioanni-Vincentelli, "Optimal Functional Decomposition Using Encoding," *Proc.* 31^{st} *DAC*, pp. 408-414, June. 1994.

[9] Francis, R. J., J. Rose, and Z. Vranesic, "Chortle-crf : Fast Technology Mapping for Lookup Table-Based FPGAs," *Proc. 28th ACM/IEEE DAC*, pp. 613-619, 1991.

[10] Francis, R. J., J. Rose, and Z. Vranesic, "Technology Mapping of Lookup Table Based FPGAs for Performance," *ACM/IEEE ICCAD*, pp. 568-571, 1991.

[11] Chen, K. C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar, "DAG-MAP: Graph-Based FPGA Technology Mapping for Delay Optimization," *IEEE Design & Test of Computers*, pp. 7-20, Sept. 1992.

[12] J. Cong, and Y. Ding, "Flowmap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup Table Based FPGA Designs," *IEEE Trans. on Computers-Aided Design of Integrated Circuits and Systems CAD*, vol. 13, pp. 1-12, Jan, 1994.

[13] Christian Legl, Bernd Wurth, and Klaus Eckl, "A Boolean Approach to Performance-Directed Technology Mapping for LUT-Based FPGA Designs," *Proc. 33th ACM/IEEE DAC*, pp. 730-735, 1996.

[14] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. on CAD*, pp. 1062-1081, Nov. 1987.

[15] R. L. Ashenhurt, "The Decomposition of Switching Functions," *Ann. Computation Lab. of Harvard Univ.*, vol. 29, pp. 74-116, 1959.

[16] R. L. Sandra Bouchard, and Alain Diou, "Methods of Logical Functions Decomposition for LUT-based FPGA," *Ann. Computation Lab. of Harvard Univ.*, vol. 29, pp. 74-116, 1959.

[17] Bhat, N. and D. Hill, "Routable Technology Mapping for FPGAs," *First Int'l ACM/SIGDA Workshop on Field Programmable Gate Arrays* , pp. 143-148, 1992.

[18] Schlag, M., J. Kong, and P. K. Chan, "Routability-Driven Technology Mapping for Lookup Table-based FPGAs," *Proc. IEEE International Conference on Computer Design*, Oct. 1992.

[19] Xilinx, *The programmable Gate Array Data Book*, Xilinx, San Jose, 1989.