Compatible Class Encoding in Hyper-Function Decomposition for FPGA Synthesis

Jie-Hong R. Jiang Jing-Yang Jou Juinn-Dar Huang

Department of Electronics Engineering
National Chiao Tung University
Hsinchu 300, Taiwan, R.O.C.

jiang@athena.ee.nctu.edu.tw jyjou@bestmap.ee.nctu.edu.tw jdhuang@ee.nctu.edu.tw

ABSTRACT

Recently, functional decomposition has been adopted for LUT based FPGA technology mapping with good results. In this paper, we propose a novel method for functional multiple-output decomposition. We first address a compatible class encoding method to minimize the compatible classes in the image function. After the encoding algorithm is applied, the decomposability will be improved in the subsequent decomposition of the image function. The above encoding algorithm is then extended to encode multiple-output functions through the construction of a hyper-function. Common sub-expressions among these multiple-output functions can be extracted during the decomposition of the hyper-function. Therefore, we can handle the multiple-output decomposition in the same manner as the single-output decomposition. Experimental results show that our algorithms are very promising.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGA's) can provide programmability for users to implement their own logic in a short turn-around time. FPGA's become increasingly popular in rapid system prototyping. Look-up table (LUT) based architecture is a prevalent one among many FPGA architectures. LUT-based FPGA's consist of an array of LUT's, each of which can implement any Boolean function with up to k (typically 4 or 5) inputs. A Boolean network can be directly realized by a one-to-one mapping between nodes and LUT's if every node in the network has up to k supports.

Functional decomposition [13,14] is a pivotal decomposition technique for LUT-based FPGA logic synthesis. Three interesting problems in functional decomposition should be noticed:

1. How to select the bound set variables?
2. How to encode the compatible classes?
3. How to extract adequate sub-expressions among multiple-output functions?

Algorithms proposed in [1,2] provide solutions to choose good bound set variables. On the other hand, approaches suggested in [3-7] deal with the second problem. These encoding algorithms can be classified into three classes according to their objectives. The first kind of encoding algorithm such as [3] simplifies the **image function** (g-function). The second kind such as [4,5] makes some **decomposition functions** (α-functions) be able to be shared among multiple-output functions. The third kind such as [6,7] minimizes the supports of the decomposition functions. As Problem 3 is considered, approaches proposed in [4,5,8] extract common sub-expressions among multiple-output functions. Extracting common decomposition functions via compatible class encoding was suggested in [4,5], while the authors of [8] tried to resubstitute decomposition functions into other functions to reduce their supports.

In this paper, we adopt the variable partitioning algorithm proposed in [2], which takes advantage of Binary Decision Diagrams (BDD's) [4,10] to conduct functional decomposition, to solve Problem 1. We will thus focus on solving Problems 2 and 3 in this paper. A new encoding algorithm is proposed to simplify the image function. Instead of reducing the number of cubes or literals in the image function as suggested in [3], our encoding algorithm aims at reducing the compatible class count. To deal with Problem 3, we transform multiple-output functions into a single-output function by introducing the hyper-function approach. Consequently, multiple-output decomposition can be reduced to an equivalent single-output decomposition. It unifies the solutions of single-output and multiple-output decomposition. Besides, the extracted common sub-expression can be large.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries. In Section 3, the compatible class encoding algorithm is proposed. Hyper-function decomposition is then discussed in Section 4. After experimental results are shown in Section 5, concluding remarks will be given in Section 6.

2. PRELIMINARIES

Let $B = \{0,1\}$. A single-output function f with n input variables $b_0,\ldots,b_{n-1}$ is denoted as $f : B^n \rightarrow B$. A function $f(b_0,\ldots,b_{n-1})$, is **decomposable** if it can be represented by another function $g(\vec{\alpha}(b_0,\ldots,b_{i-1}),b_j,\ldots,b_{n-1}) = g(\alpha_0(b_0,\ldots,b_{i-1}),\ldots,\alpha_{t-1}(b_0,\ldots,b_{i-1}), b_j,\ldots,b_{n-1})$, where $0 < t < j \leq i$. The decomposition is disjoint if $j = i$. When $j = i$, $\{b_0,\ldots,b_{i-1}\}$ is called the **bound (λ) set** and $\{b_j,\ldots,b_{n-1}\}$ is called the **free (μ) set**. In this paper, only the disjoint decomposition is considered.

Definition 2.1 Let X and Y be two sets of binary variables, $X \cap Y = \emptyset$. Given a completely specified function f: $B^{|X|} \times B^{|Y|} \rightarrow B$, with X being the λ set and Y being the μ set. We say that $\mathbf{x}_1, \mathbf{x}_2 \in B^{|X|}$ are **compatible** with respect to f, denoted as $\mathbf{x}_1 \sim \mathbf{x}_2$, if $\forall$ $y \in B^{|Y|}$, $(\mathbf{x}_1,y)$ and $(\mathbf{x}_2,y) \in B^{|X|} \times B^{|Y|}$ such that $f(\mathbf{x}_1,y) = f(\mathbf{x}_2,y)$.❑

All mutually compatible elements form a **compatible class**.

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50 DAC98 - 06/98 San Francisco, CA USA

Theorem 2.1 $\forall (\mathbf{x},\mathbf{y}) \in B^{|X|}\times B^{|Y|}$, f: $B^{|X|}\times B^{|Y|}\rightarrow B$, $\vec{\alpha}$: $B^{|X|}\rightarrow W$, g: $W\times B^{|Y|}\rightarrow B$,

$$f(\mathbf{x},\mathbf{y}) = g(\vec{\alpha}(\mathbf{x}),\mathbf{y}) \quad (1)$$

holds if and only if

$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in B^{|X|},\ \vec{\alpha}(\mathbf{x}_1) = \vec{\alpha}(\mathbf{x}_2) \Rightarrow \mathbf{x}_1 \sim \mathbf{x}_2 \quad (2)$$ ❑

$\vec{\alpha}$ is a function with binary inputs and a symbolic output. The number of the admissible values in W, $|W|$, must be not less than the number of compatible classes. To implement $\vec{\alpha}$ by binary logic, at least $t = \lceil \log_2|W| \rceil$ binary functions, $\alpha_0,..., \alpha_{t-1}$, are required to encode $\vec{\alpha}$. When $t = \lceil \log_2|W| \rceil$, we say that this encoding is **rigid**. Otherwise when $t > \lceil \log_2|W| \rceil$, the encoding is **pliable**. Eq. (1) can be rewritten as $f(\mathbf{x},\mathbf{y}) = g(\alpha_0(\mathbf{x}),...,\alpha_{t-1}(\mathbf{x}),\mathbf{y})$. For a single-output function, if each compatible class is assigned just one code, then this encoding is **strict** and Eq. (2) can be redefined as

$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in B^{|X|},\ \vec{\alpha}(\mathbf{x}_1) = \vec{\alpha}(\mathbf{x}_2) \Leftrightarrow \mathbf{x}_1 \sim \mathbf{x}_2.$$

In contrast, if there exists any compatible class encoded with more than one code, then the encoding is **non-strict**.

3. COMPATIBLE CLASS MINIMIZATION

Two important factors affect the decomposition quality: one is the variable partitioning and the other is the compatible class encoding. We solve the variable partitioning problem by using the algorithm proposed in [2]. Moreover, we intend to encode compatible classes to reduce the number of compatible classes in the next decomposition of the image function. Before our discussing the encoding technique in Subsection 3.2, don't care assignment that is used in the encoding is introduced first in Subsection 3.1.

3.1 Don't Care Assignment

The authors of [8] used the don't care assignment to minimize the supports of an incompletely specified function. However, we formulate the don't care assignment as the clique partitioning problem in order to reduce the number of compatible classes instead of minimizing the number of supports.

We record the compatible relationship among λ set vertices by using the compatibility graph. Each λ set vertex corresponds to a vertex in the graph. A pair of vertices are connected by an edge if and only if these two vertices cab be compatible under certain don't care assignments. After constructing the graph, we want to find the least number of cliques such that each vertex is covered by exactly one clique; thus the number of cliques equals to the number of compatible classes. Because the clique partitioning problem is NP-complete, we adopt the heuristics appeared in [9] to have a polynomial time solution.

3.2 Compatible Class Encoding

The authors of [3] assumed that the fewer cubes or literals in the image function, the better decomposition quality could be obtained. Hence, the compatible class encoding problem was modeled as the symbolic-input encoding problem to minimize the number of cubes or literals of the image function. However, those counts may not be a good cost function for LUT-based FPGA synthesis. In this paper, we formulate the encoding problem as minimizing the number of compatible classes generated at the subsequent decomposition of the image function. The new cost function has better meaning for LUT architecture.

After λ set selection and don't care assignment, compatible classes are fixed. The next step is to encode these compatible classes. In order to exploit more don't care set, we take the strict encoding policy. Example 3.1 illustrates why an encoding is relevant to the number of compatible classes in the decomposition of the image function.

Example 3.1 Assume that the targeted LUT can implement any 4-input functions. We want to decompose the function f in Figure 1(a) with $\{a,b,c\}$ as the λ set selection.

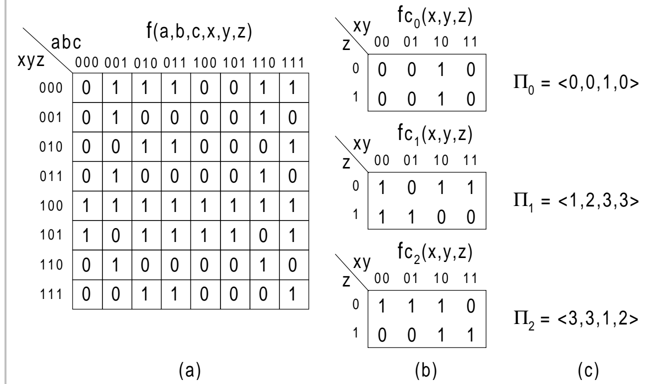

Figure 1: (a)Decomposition Chart of f (b)Compatible Class Functions (c)Symbolic Notations of Column Patterns

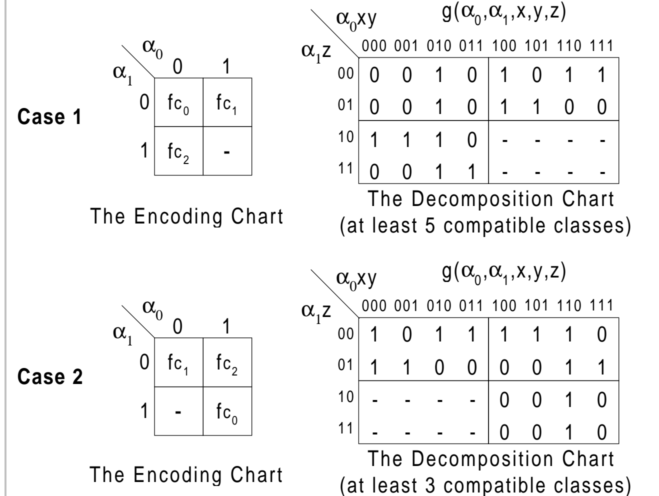

Figure 2: Encoding Chart and Decomposition Chart

In the decomposition of f if a, b and c are selected as the λ set variables, there are three compatible classes with functions as shown in Figure 1(b). Two α-functions, $\alpha_0(a,b,c)$ and $\alpha_1(a,b,c)$, are needed to encode three compatible classes. Suppose we choose α_0, x, and y as the λ set variables in the decomposition of $g(\alpha_0,\alpha_1,x,y,z)$. Examining the two encoding cases in Figure 2, we can see that the encoding may affect the number of compatible classes in the decomposition of the g-function. Note that "-" represents don't care.

Theorem 3.1 The encoding does not affect the number of compatible classes in the subsequent decomposition of the image function if all of the α-functions are selected together in the λ set or in the μ set of the subsequent decomposition of the image function. ❑

Theorem 3.2 After the λ set variables used in the decomposition of an image function have been selected, to reduce the number of compatible classes of the image function, we only have to determine which compatible class functions should belong to the same column or the same row in the encoding chart. The exact codes of these columns and rows do not influence the number of compatible classes of the image function. ❑

In the rest of this paper, we use symbolic notations (decimal numbers) to represent column patterns.

Definition 3.1 A **partition Π**, $<s_0,..,s_{n-1}>$, is a symbolic notation of n column patterns. Elements s_i equals to s_j if and only if i^{th} column pattern equals to j^{th} column pattern. ❑

For example in Figure 1, Π_0, Π_1 and Π_2 in (c) are symbolic notations of column patterns in the charts of fc_0, fc_1 and fc_2 in (b) respectively. A **conjunction partition Πc** (**disjunction partition Πd**) of a set of partitions is a new partition which is a symbolic notation of column patterns formed by stacking these partitions vertically in the same column (horizontally in the same row) of the encoding chart. The **multiplicity** of a partition is the number of different symbols in this partition.

```
Algorithm: Encoding
Input:     Compatible Class Functions fc's
Output:    Image Function with its λ Set Variables
begin
1    g′ ← Encode compatible classes at random;
2    if ( g′ is κ–feasible )   return {g′,∅};
3    {λ′,#compatible_classes} ← Variable_Partitioning (g′);
     /*From g′,λ′ and fc's, we can derive the # of rows, #R, the # of
     columns, #C, in the encoding chart and the partitions of fc's, Π_fc's.*/
4    if ( #R==1 or #C==1 ) return {g′,λ′};      /*According to Theorem 3.1*/
     /*Each Π_fc occupies a distinct row set and a distinct column set initially.*/
5    CSet's ← CombineColumnSets (Π_fc's);
6    while ( |RSet's| > #R or |CSet's| > #C )
7         {RSet's,CSet's} ← CombineRowSets (RSet's,CSet's);
8    if ( random encoding has less compatible classes )   return {g′,λ′};
9    return ( {g,λ′} ← Encoding according to RSet's and CSet's );
end
```

Figure 3: The Encoding Procedure

Partition	Positions with the same content
Π_2	p0p3
Π_3	p1p3
Π_4	p1p3
Π_5	p0p2
Π_6	p1p2p3
Π_7	p0p1p3
Π_8	p0p2, p1p3

(a)

Partition	Positions with the same content
Πc of $\{\Pi_2,\Pi_7\}$	p0p3
Πc of $\{\Pi_3,\Pi_4,\Pi_6,\Pi_7,\Pi_8\}$	p1p3
Πc of $\{\Pi_5,\Pi_8\}$	p0p2

(b)

Figure 4: Partitions and Positions with the Same Contents

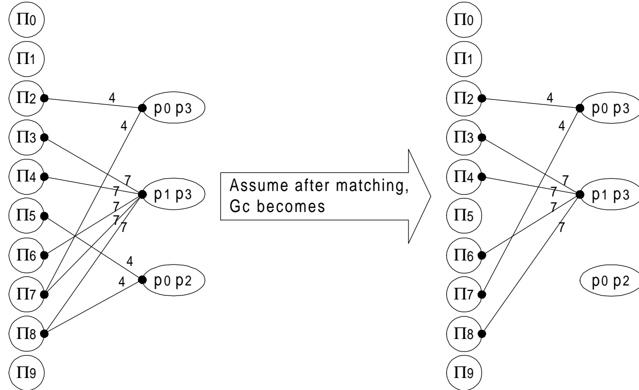

Figure 5: Graph Formulation of Column-Set Combination

The encoding draft is shown in Figure 3. We explain Steps 5 and 7 in detail with Example 3.2.

Example 3.2 Assume we have ten compatible class functions, fc_0,..., fc_9, with their partitions Π_0,..., Π_9 respectively as follows.

Π_0=<0,1,2,3> Π_1=<0,2,1,3> Π_2=<3,0,1,3> Π_3=<2,1,0,1> Π_4=<0,1,3,1> Π_5=<0,1,0,2> Π_6=<1,0,0,0> Π_7=<1,1,2,1> Π_8=<1,2,1,2> Π_9=<3,2,1,0>

Suppose we want to place these partitions (or these compatible class functions) in the encoding chart with #R = 4 and #C = 4.

Step 5 in Figure 3: *Evaluate which compatible classes should be bounded in the same column of the encoding chart*.

We represent position i in a partition as p_i for convenience. Since the contents of p_1 and p_3 in Π_4 are the same, we say that positions with the same content of Π_4 is p_1p_3. For the above ten partitions, Figure 4(a) records the information. Positions with the same content such as p_0p_3, we denote it as $\boldsymbol{Psc_{03}}$. $|Psc_{03}| = 2$ because there are two positions, p_0 and p_3, in Psc_{03}. As there are two partitions Π_2 and Π_7 having Psc_{03}, ***#Partitions***$(Psc_{03}) = 2$ and ***Partitions***$(Psc_{03}) = \{\Pi_2,\Pi_7\}$. Therefore, the conjunction partition of Π_2 and Π_7 has the same content in p_0 and p_3. For all Psc's in Figure 4(a) (Psc_{03}, Psc_{13}, Psc_{02}, Psc_{123}, or Psc_{013}) with $\#Partitions(Psc) \geq 2$, we list them in Figure 4(b). We then build a **column-graph** $Gc(Vc,Uc,Ec)$ as depicted in Figure 5, which is a bipartite graph. For each partition, there is a corresponding vertex in Vc; for each Psc in Figure 4(b), there are $\lceil(\#Partitions(Psc)-1)/\#R\rceil$ corresponding vertices, u_{Psc}'s, in Uc. (It is because that $Partitions(Psc)$ may be collected in more than one column set.) A vertex in Uc corresponding to a Psc has $\#Partitions(Psc)$ edges connecting this vertex with vertices in Vc which are corresponding to $Partitions(Psc)$. The weight of an edge connecting to u_{Psc} equals to ($|Psc|$+the number of edges connecting to u_{Psc}). We then find a *b-matching* [12], Mc, of maximum weight for Gc. For this b-matching, each vertex in Vc is connected by at most one edge; each vertex in Uc is connected by at most #R edges. After matching, the corresponding partitions of vertices $\in Vc$ connected with the same u_{Psc} are grouped in the same column set. According to the matching result in Figure 5, we have six column sets, $\{\Pi_3,\Pi_4,\Pi_6,\Pi_8\}$, $\{\Pi_2,\Pi_7\}$, $\{\Pi_0\}$, $\{\Pi_1\}$, $\{\Pi_5\}$ and $\{\Pi_9\}$. After applying the above algorithm, we can reduce the multiplicity of the Πc of partitions in the same column set.

Step 7 in Figure 3: *Evaluate which compatible classes should be bounded in the same row and in adjacent columns of the encoding chart*.

Assume each partition initially occupies a separate row set. We calculate the benefits of merging pairs of row sets. For Π_i and Π_j in different column sets, we calculate the benefit of merging them as $\sigma\times Br_{ij}+\tau\times Bc_{ij}$; otherwise their merging benefit is calculated as $\sigma\times Br_{ij}+\tau\times Bc_{ij}$–the weight of the edge connecting the vertex corresponding to Π_i in Vc of Gc. Because when Π_i and Π_j are in the same column set, we don't want to tear them into different columns. The detailed calculations of σ, Br_{ij}, τ, and Bc_{ij} are as follows.

$Br_{ij} = n-(n_{ij}-n_i)-(n_{ij}-n_j)$

$$Bc_{ij} = \sum_{\text{every symbol } \mathbf{S} \text{ in all } \Pi\text{'s}} ((\text{the \# of } \mathbf{S} \text{ in } \Pi_i \text{ and } \Pi_j) - k)^2$$

σ = (# of row sets so far) – #R (if $\sigma < 0 \Rightarrow \sigma = 0$)

τ = (# of column sets so far) – #C (if $\tau < 0 \Rightarrow \tau = 0$)

$k = m/n$

m: There are m positions in the Πd of Π_i and Π_j.

n: There are totally n kinds of symbols in all partitions.

n_{ij}: There are totally n_{ij} kinds of symbols in the Πd of Π_i and Π_j.

n_i, n_j: There are n_i and n_j kinds of symbols in Π_i and Π_j respectively.

According to the calculated benefits, we construct a **row-graph** $Gr(Vr,Er)$. Each partition Π_i has a corresponding vertex $v_i \in Vr$; each pair of vertices (v_i,v_j) is connected by an undirected edge whose weight is the benefit of merging Π_i and Π_j in the same row set. We then find the *maximum-cardinality matching* [12], Mr, of Gr. For each edge $\in Mr$, the corresponding partitions of its two end vertices are hopefully to be combined together in a row set. We

combine these pairs of partitions iteratively with benefits from high to low until the number of current row sets is not greater than #R or all edges $\in$ *Mr* have been selected. In this example, $\{\Pi_7,\Pi_8\}$, $\{\Pi_5,\Pi_6\}$, $\{\Pi_2,\Pi_4\}$, $\{\Pi_0,\Pi_9\}$ and $\{\Pi_1,\Pi_3\}$ are therefore selected in succession. According to column sets derived in Step 5, we stack these pairs of partitions properly. If there are some conflicts between Step 5 and Step 7, we assume that the decisions of Step 7 have higher priority than those of Step 5. So far we have 5 row sets $\{\Pi_7,\Pi_8\}$, $\{\Pi_5,\Pi_6\}$, $\{\Pi_2,\Pi_4\}$, $\{\Pi_0,\Pi_9\}$, $\{\Pi_1,\Pi_3\}$ and 4 column sets $\{\Pi_3,\Pi_4,\Pi_6,\Pi_8\}$, $\{\Pi_1,\Pi_2,\Pi_5,\Pi_7\}$, $\{\Pi_0\}$, $\{\Pi_9\}$. We have row-column relation as illustrated in Figure 6(a).

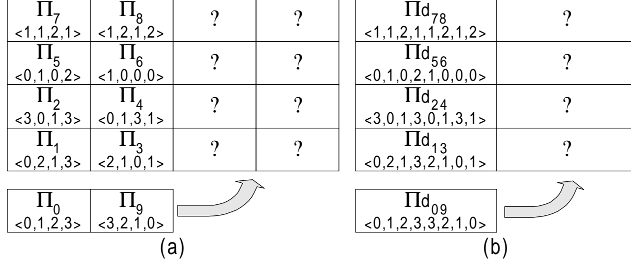

Figure 6: Row-Column Relation

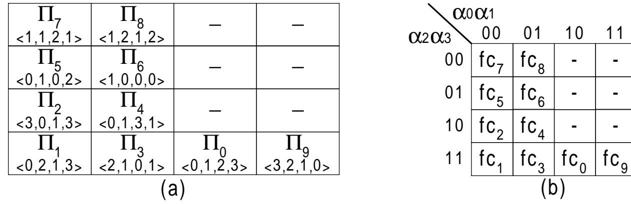

Figure 7: (a)Final Row-Column Relation (b)Final Encoding

Step 7 must be repeated until the number of row sets $\leq$ #R and the number of column sets $\leq$ #C. In this example so far, since there are 5 row sets, 5 > #R, we thus iterate Step 7 to calculate the benefits of combining pairs of row sets. For each row set, we represent it by the disjunction partition of the partitions in this row set as shown in Figure 6(b). After constructing the new *Gr'*(*Vr'*,*Er'*) and finding the *Mr'* for it, we combine the pair of row sets with maximum benefits ($\{\Pi_1,\Pi_3\}$,$\{\Pi_0,\Pi_9\}$) into a row set $\{\Pi_1,\Pi_3,\Pi_0,\Pi_9\}$. Since the number of row sets becomes 4 (< #R), we don't have to combine row sets further. Finally as shown in Figure 7(a), we have 4 row sets and 4 column sets. According to Theorem 3.2, we know that the codes of columns and rows do not affect the number of compatible classes in the decomposition of the image function. So we can encode these compatible classes as that indicated in Figure 7(b). After encoding, we can derive the real image function. In the decomposition of this image function, we will select α_0, α_1 and some original bound set variables in the previous decomposition as the bound set variables according to Step 3 in Figure 3. Consequently, we will have 4 compatible classes in the decomposition of the image function.

4. HYPER-FUNCTION DECOMPOSITION

To solve multiple-output decomposition in the same manner as single-output decomposition, we propose a solution, which is called hyper-function decomposition.

Definition 4.1 A set of distinct Boolean functions $\{f_0,\ldots,f_{n-1}\}$, called **ingredients**, can be combined together to form a single-output **hyper-function** by using additional $i = \lceil \log_2 n \rceil$ binary bits $\eta_0,\ldots,\eta_{i-1}$, called **pseudo primary inputs**, to encode these ingredients. The supports of a hyper-function include pseudo primary inputs and the union of supports of its ingredients. ❑

Hyper-function transforms multiple-output functions into a single-output function. It can perform the function of any individual ingredient by assigning the corresponding code to the pseudo primary inputs. In Subsection 4.1, we discuss how to encode the ingredients to form a hyper-function with better decomposability.

4.1 Encoding of Ingredients

Actually, a hyper-function can be considered as an image function and its ingredients can be viewed as compatible class functions. Consequently, Theorem 3.1 and Theorem 3.2 can be extended as follows.

Theorem 4.1 The encoding does not affect the number of compatible classes in the subsequent decomposition of the hyper-function if all of the pseudo primary inputs are selected together in the λ set or in the μ set of the subsequent decomposition of the hyper-function. ❑

Theorem 4.2 After the λ set variables used in the decomposition of a hyper-function have been selected, to reduce the number of compatible classes of the hyper-function, we only have to determine which ingredients should belong to the same column or the same row in the encoding chart. The exact codes of these columns and rows do not influence the number of compatible classes of the hyper-function. ❑

We thus use the same encoding strategy to encode these ingredients as that used in the compatible class encoding.

4.2 The Decomposition of Hyper-Function

Single-output decomposition can be easily applied to the decomposition of a hyper-function. Via hyper-function decomposition, any algorithm that is proposed for single-output decomposition can be applied to multiple-output decomposition.

Definition 4.2 The **transitive fanout** of a node *j*, denoted as ***TFOj***, is defined as $TFOj = \{\text{node } i \mid i = j \text{ or } \exists \text{ path from } j \text{ to } i\}$.❑

Definition 4.3 The **duplication source** (***DS***) after a hyper-function decomposition is the set of nodes which have at least one pseudo primary input as their direct fanin. ❑

Note that after we have decomposed a hyper-function, every node $\notin DS$ must be *k*-feasible. However, every node $\in DS$ with *t* pseudo primary inputs as its direct fanins must be (*t*+*k*)-feasible.

Definition 4.4 The **duplication cone** (***DC***) after a hyper-function decomposition is defined as $DC = \{\cup_j TFOj \mid j \in DS\}$. ❑

Definition 4.5 The $\boldsymbol{m^{th}}$ **layer duplication set** ($\boldsymbol{DSet_m}$) after a hyper-function decomposition is defined as $DSet_m = \{\text{node } j \mid j \text{ is in } TFO\text{'s of } m \text{ pseudo primary inputs}\}$. ❑

After we have decomposed a hyper-function, the *DC* should be duplicated to implement the ingredients. Assume that the hyper-function has *n* pseudo primary inputs and *i* ingredients. A node $\in DSet_m$ ($m < n$) must be duplicated ($2^m - 1$) additional copies; a node $\in DSet_n$ must be duplicated to have additional ($i - 1$) copies. To implement each ingredient, we then assign its corresponding code to the pseudo primary inputs. These pseudo primary inputs, assigned with constant values, can be collapsed into their fanout nodes. Nodes $\in$ *DS* are thus reduced by eliminating the extra pseudo primary inputs. After a hyper-function decomposition, all new generated nodes $\notin$ *DC* can be shared by these ingredients.

Example 4.1 Assume that four distinct Boolean functions, $f_0(i_0,i_1,i_2,i_3,i_4,i_5,i_7,i_8)$, $f_1(i_0,i_1,i_2,i_3,i_4,i_5,i_6)$, $f_2(i_0,i_1,i_2,i_3,i_4,i_5)$ and $f_3(i_0,i_1,i_2,i_3,i_4,i_5)$, form a hyper-function F: $B^{11} \rightarrow B$. Suppose the coding of each ingredient is derived by applying our compatible classes encoding algorithm and is shown in Figure 8(a). To imple-

ment F with 5-input LUT's, assume that F is decomposed as what is depicted in Figure 8(b), in which nodes ∈ duplication cone are filled with gray. After duplicating the duplication cone, as demonstrated in Figure 9(a), we assign (0,0) to (η_0,η_1) to recover f_0, (1,0) to recover f_1, (0,1) to recover f_2 and (1,1) to recover f_3.

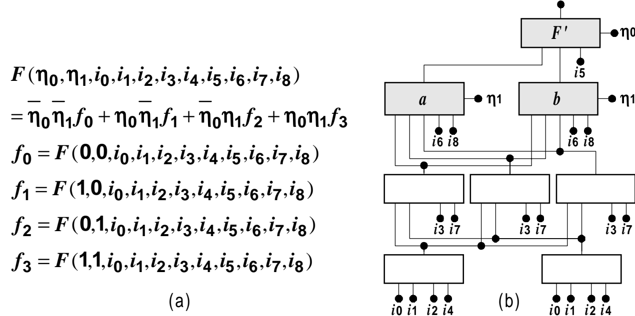

Figure 8: (a)Hyper-Function and Ingredients (b)Decomposition of F

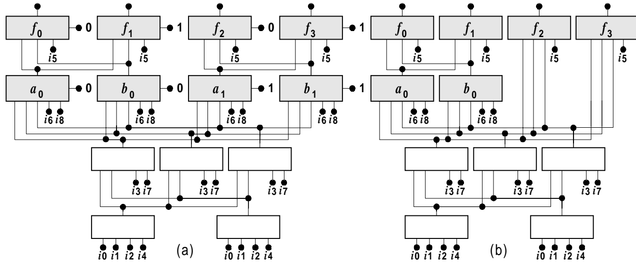

Figure 9: (a)Duplication for Ingredients (b)Further Reductions

After our collapsing these constant input signals in Figure 9(a) into their fanout nodes, the resultant network as is indicated in Figure 9(b) will be independent of these pseudo primary inputs. Since f_2 and f_3 are independent of i_6, i_7 and i_8 according to their original function expressions, we can further simplify the network by collapsing nodes a_1 and b_1 into f_2 and f_3 as shown in Figure 9(b). Nodes ∉ duplication cone can be shared by the ingredients of F.

4.3 Properties of Hyper-Function Decomposition

Because nodes in the duplication cone must be duplicated, fewer nodes in the cone are preferred. Consequently, we should keep the pseudo primary inputs as close to the output as possible during the decomposition process. In other words, pseudo primary inputs are preferred to be kept in the μ set during decomposition. In the extreme case, if we always select the pseudo primary inputs in the μ set, hyper-function decomposition can be considered as the *column encoding method* in [4]. Hence, the column encoding approach in [4] is a special case of our hyper-function decomposition. Hyper-function decomposition provides a more generic and flexible means to extract common sub-logic than column encoding. Since multiple-output functions can be decomposed as easily as a single-output function, the λ set size doesn't have to be restricted to a small value. Moreover, the shared logic can cross many levels; the extracted common sub-expressions can be large.

Although a hyper-function is decomposed by applying single-output decomposition, two differences exist between hyper-function decomposition and single-output decomposition. First, strict encoding for a single-output decomposition may become non-strict for a hyper-function decomposition. Because for each ingredient of the hyper-function, a compatible class can be encoded with more than one code. (When a hyper-function is constructed, conjunction partitions may be performed on the partitions of ingredients. Hence a pattern in a partition may be broken into several patterns in a conjunction partition.) Secondly, an encoding that is rigid for a single-output decomposition may become pliable for a hyper-function decomposition. Because the compatible classes of an ingredient may be encoded by more bits than necessary. IMODEC [5] provides a good approach to guarantee that each function is rigidly encoded. However, pliable encoding can save more areas than rigid one in the cases discussed below.

Definition 4.6 A partition **A** is **contain**ed by another partition **B** if the multiplicity of **B** equals to the multiplicity of the conjunction partition of {**A**,**B**}. ❑

Theorem 4.3 Given two partitions **A** of function f_a and **B** of function f_b with respect to the same λ set selection and both of $\lceil \log_2 |\text{multiplicity of } \mathbf{A}| \rceil$ and $\lceil \log_2 |\text{multiplicity of } \mathbf{B}| \rceil$ are less than the λ set size. **A** is contained by **B** if and only if the decomposition functions of f_b (which identify the column patterns in **B** by strict encoding) can be used as the decomposition functions of f_a. ❑

Theorem 4.4 Given two partitions **A** of function f_a and **B** of function f_b with respect to the same λ set selection and both of $\lceil \log_2 |\text{multiplicity of } \mathbf{A}| \rceil$ and $\lceil \log_2 |\text{multiplicity of } \mathbf{B}| \rceil$ are less than the λ set size. If **A** is contained by **B**, then the decomposition functions of f_b can be used as the decomposition functions of f_a. ❑

Example 4.2 Given three functions $f_0(x_0,x_1,x_2,x_3,y_0,y_1)$, $f_1(x_0,x_1,x_2,x_3,y_2,y_3)$ and $f_2(x_0,x_1,x_2,x_3,y_3,y_4)$ with λ set selection as $\{x_0,x_1,x_2,x_3\}$, assume therefore we have three partitions:

Π_0 = <0,0,1,0,1,2,2,0,3,2,0,0,0,0,0,2> of f_0,

Π_1 = <0,1,2,0,2,3,3,2,4,3,0,2,1,5,1,3> of f_1, and

Π_2 = <0,1,1,0,1,2,2,3,3,2,0,3,1,4,5,2> of f_2.

If f_1 and f_2 are combined to construct a hyper-function h_{12} with λ set $\{x_0,x_1,x_2,x_3\}$, then the hyper-function has partition Πc of $\{\Pi_1,\Pi_2\}$, Πc_{12}. Because Πc of $\{\Pi_0,\Pi_1,\Pi_2\}$, Πc_{012}, has the same multiplicity as Πc_{12}, Π_0 is contained by Πc_{12} by Definition 4.6. According to Theorem 4.3 or Theorem 4.4, the decomposition functions of h_{12} can be used as the decomposition functions of f_0. Therefore, if f_0, f_1 and f_2 are combined to form a hyper-function with λ set $\{x_0,x_1,x_2,x_3\}$, there are three decomposition functions (because of the multiplicity of Πc_{012} = 8) shared by the three functions as shown in Figure 10(a). Because f_0 uses three decomposition functions instead of two decomposition functions to encode four compatible classes, the encoding becomes pliable. On the other hand, if the encoding is restricted to being rigid, such as [5], it may derive the result as shown in Figure 10(b). In this case, two more LUT's are consumed.

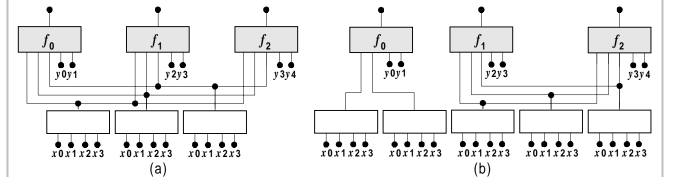

Figure 10: (a)Pliable (b)Rigid Encoding of f_0

5. EXPERIMENTAL RESULTS

Our algorithm, HYDE, has been implemented in SIS environment [11]. Experiments are conducted over a set of benchmark circuits. To prepare the initial circuits for the following technology mapping, small circuits are collapsed, while large circuits are optimized by applying SIS algebraic script. (The benchmark circuit **des** is, in addition, partially collapsed such that several nodes can share the same supports.) After the initial circuits are derived, the technology mapping script used for 2-level circuits is: our decomposition, xl_partition -tm, xl_cover and the script for multi-level circuits is: (full_)simplify, our decomposition, xl_partition -tm, 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. The experiments are run under SUN SPARC 20 workstation. We compare these results with other state-of-the-art FPGA synthesis techniques [4,5,8] in Table 1 and Table 2.

Table 1: Experimental Results for XC3000 Device

Circuit	IMODEC[5] CLB	FGSyn[4] CLB	HYDE CLB	CPU Time sec.
5xp1	9	9	10	1.3
9sym	7	7	6	22.8
alu2	46	55	43	554.4
alu4	168	56	140	911.7
apex6	129	181	135	108.7
apex7	41	43	39	9.6
clip	12	18	11	407.2
count	26	23	24	1.6
des	489	-	408	236.6
duke2	122	85	75	28.0
e64	55	44	48	0.0
f51m	8	8	8	10.4
misex1	9	8	9	11.8
misex2	21	22	22	3.3
rd73	5	5	5	3.0
rd84	8	8	7	16.0
rot	127	136	125	132.7
sao2	17	25	17	117.5
vg2	19	17	18	3.6
z4ml	4	4	4	2.7
C499	50	54	50	2.9
C880	81	87	68	69.8
Total	**1453**		**1272**	
Subtotal	**964**	**895**	**864**	

Table 2: Experimental Results for 5-input 1-output LUT's

Circuit	without re-sub. [8] LUT	with resub. [8] LUT	resub. PO[8] LUT	HYDE LUT
5xp1	15	11	10	13
9sym	7	7	7	6
alu2	48	48	48	50
alu4	172	90	56	206
apex4	374	374	374	354
apex6	192	161	155	186
apex7	120	61	54	54
b9	53	39	37	36
clip	18	11	14	14
count	52	31	31	31
des	-	-	-	561
duke2	175	155	150	116
e64	-	-	-	80
f51m	12	10	8	12
misex1	12	10	10	13
misex2	40	36	36	29
misex3	195	213	120	131
rd73	8	6	6	6
rd84	12	7	8	9
rot	-	-	-	185
sao2	23	21	21	22
vg2	44	21	17	18
z4ml	6	5	4	5
C499	-	-	-	70
C880	-	-	-	81
Total	**1578**	**1317**	**1166**	**1311**
Subtotal(-alu4)	**1406**	**1227**	**1110**	**1105**

In Table 1, the target architecture is the Xilinx XC3000 FPGA. Our algorithm is compared with IMODEC [5] and FGSyn [4]. In Table 2, the resultant circuits are constructed by 5-input 1-output LUT. Without much difference in the consumed CPU time, it is not shown in Table 2. In column 2, 3 and 4 of Table 2, we repeat the results reported in [8]. Although not mapping **alu4** well, HYDE still outperforms IMODEC and FGSyn. Excluding **alu4**, our algorithm produces slightly better results than those in column 4 of Table 2. Due to the disability of handling large circuits such as **C880** in [8], our algorithm is considered more practical to handle large circuits.

6. CONCLUSIONS

Compatible class encoding and hyper-function decomposition techniques have been proposed. The former improves the decomposability of the image function and the hyper-function, while the latter extracts common sub-expressions among multiple-output functions. By transforming these multiple-output functions into a single-output hyper-function, the problem of multiple-output decomposition can thus be reduced to that of single-output decomposition. As a result, previous algorithms intended for single-output decomposition can be easily extended to solve multiple-output decomposition. Experimental results show that our approach is practical and promising. We believe that hyper-function decomposition can be not only used for FPGA synthesis but also exploited to identify common sub-logic in the technology-independent optimization phase of logic synthesis. Another possibility of application is the time-multiplexed reconfigurable computing. For time-multiplexed functions, we can combine them together as a hyper-function. After decomposition, we don't have to duplicate the duplication cone at all. Instead, we can use the pseudo primary inputs to recover the time-multiplexed functions. More applications and theoretical works need to be explored in the future.

REFERENCES

[1] Wen-Zen Shen, Juinn-Dar Huang and Shih-Min Chao, "Lambda Set Selection in Roth-Karp Decomposition for LUT-Based FPGA Technology Mapping," *Proc. 32nd DAC*, pp.65-69, June 1995.

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

[3] Rajeev Murgai, Robert K. Brayton and Alberto Sangiovanni-Vincentelli, "Optimum Functional Decomposition Using Encoding," *Proc. 31st DAC*, pp.408-414, June 1994.

[4] Yung-Te Lai, Kuo-Rueih Ricky Pan and Massoud Pedram, "OBDD-Based Function Decomposition: Algorithms and Implementation," *IEEE Trans. CAD*, vol. 15, pp.977-990, August 1996.

[5] Bernd Wurth, Klaus Eckl and Kurt Antreich, "Functional Multiple-Output Decomposition: Theory and an Implicit Algorithm," *Proc. 32nd DAC*, pp.54-59, June 1995.

[6] Juinn-Dar Huang, Jing-Yang Jou and Wen-Zen Shen, "Compatible Class Encoding in Roth-Karp Decomposition for Two-Output LUT Architecture," *Proc. Intl. Conf. on CAD*, pp.359-363, Nov. 1995.

[7] Christian Legl, Bernd Wurth and Klaus Eckl, "An Implicit Algorithm for Support Minimization during Functional Decomposition," *Proc. European Design and Test Conf.*, 1996.

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

[9] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu and Steve Y-L Lin, *High-Level Synthesis*, Kluwer Ac. Pub., 1992.

[10] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, pp.677-691, August 1986.

[11] 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.

[12] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley, New York, 1988.

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

[14] J. P. Roth and R. M. Karp, "Minimization Over Boolean Graphs," *IBM Journal*, pp.227-238, 1962.