# Computing Support-Minimal Subfunctions During Functional Decomposition

Christian Legl, Bernd Wurth, and Klaus Eckl

*Abstract*— **The growing popularity of look-up table (LUT)-based field programmable gate arrays (FPGA's) has renewed the interest in functional or Roth–Karp decomposition techniques. Functional decomposition is a powerful decomposition method that breaks a Boolean function into a set of subfunctions and a composition function. Little attention has so far been given to the problem of selecting good subfunctions after partitioning the input variables into the disjoint bound and free sets. Therefore, the extracted subfunctions usually depend on all bound variables. In this paper,[1] we present a novel decomposition algorithm that computes subfunctions with a minimal number of inputs. This reduces the number of LUT's and improves the usage of multiple-output SRAM cells. The algorithm iteratively computes subfunctions; in each iteration step it implicitly computes a set of possible subfunctions and finds a subfunction with minimal support. Moreover, our technique finds nondisjoint decompositions, and thus unifies disjoint and nondisjoint decomposition. The algorithm is very fast and yields substantial reductions of the number of LUT's and SRAM cells.**

*Index Terms*—**Programmable logic array, technology mapping.**

## I. INTRODUCTION

A POPULAR class of field programmable gate arrays (FPGA's) is based on the look-up table (LUT) as the basic programmable logic block. A $k$-input look-up table, which is usually implemented by a $2^k$-bit static random-access memory (SRAM) cell, can realize any Boolean function of up to $k$ variables. Many FPGA architectures have SRAM cells with several outputs that allow an optimal use of the memory. For example, a Xilinx XC5000 FPGA is made up of 64-bit memory cells that can be used to implement either two five-input functions or four four-input functions.

A variety of techniques for logic synthesis to LUT-based FPGA's has been developed in recent years. Many of these techniques use functional decomposition [1]–[12]. Functional decomposition was pioneered in the early 1960's by Ashenhurst, Curtis, Roth, and Karp [13], [14] and is therefore sometimes called Roth–Karp decomposition. This kind of decomposition has the advantage that it always yields functions with fewer inputs than the original function. Thus it nicely reflects the constraint on the number of LUT inputs. Moreover, functional decomposition is a powerful Boolean logic synthesis method.

Functional decomposition breaks a Boolean function $f$ into the composition function $g$ and the subfunctions $d_i$ such that $f(\mathbf{x},\mathbf{y}) = g(d_1(\mathbf{x}),\cdots,d_c(\mathbf{x}),\cdots,\mathbf{y})$. The two main problems during functional decomposition are: First, how shall the input variables be partitioned into bound variables $\mathbf{x}$ and free variables $\mathbf{y}$? Given such an input partition into bound set and free set, there exists a minimum number of subfunctions $d_i$. Even with such a minimum number of subfunctions, which also minimizes the number of inputs to the composition function $g$, there are many degrees-of-freedom for the selection of the subfunctions. The second problem thus is: which subfunctions $d_i(\mathbf{x})$ shall be chosen? Note that the problem of computing subfunctions is equivalent to the problem of encoding the set of vertices of the bound variables. Once the subfunctions have been selected, the computation of the composition function is simple.

The degrees-of-freedom that exist for the selection of subfunctions are usually not exploited. One notable exception is a recent technique that exploits them to obtain composition functions with a minimal number of literals [2]. This optimization can be valuable if the composition function has more than $k$ inputs and needs to be decomposed further. Of course, literal count minimization does not guarantee an improved decomposability of a function [2].

The subfunctions $d_i(\mathbf{x})$ usually depend on all bound variables. However, each bound variable must only be an input to at least one subfunction, and not necessarily to all subfunctions. An important problem, which we address in this work, is the computation of such subfunctions $d_i$ that have minimal support. The advantages are manifold. In case of bound sets with cardinality larger than $k$, support-minimal subfunctions can have $k$ or less inputs; then, further decomposition is not necessary any more. This reduces the final LUT count and sometimes even the circuit depth. If an SRAM cell can implement several functions as described above, reducing the support allows to put more functions into a given cell. Moreover, support minimization reduces the number of connections and therefore increases routability. Since routing resources are fixed on an FPGA, improved routability is a very attractive feature.

From a theoretical point of view, it is interesting to observe that disjoint functional decomposition degenerates to nondisjoint decomposition if a subfunction depends on just one bound variable. Thus, computing support-minimal sub-

---

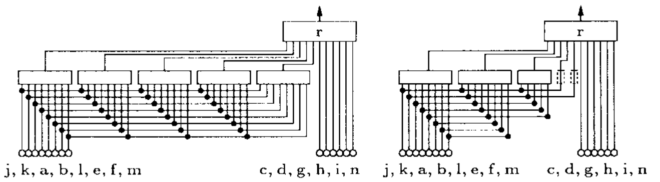
Manuscript received March 26, 1997; revised November 1, 1997.

C. Legl and K. Eckl are with the Institute of Electronic Design Automation, Technical University of Munich, Munich 80290 Germany.

B. Wurth was with Synopsys, Inc., Mountain View, CA 94043 USA. He is now with Siemens Semiconductor Group, Munich 81617 Germany.

Publisher Item Identifier S 1063-8210(98)05972-1.

[1] See the Guest Editorial of the Special Issue on Field Programmable Gate Arrays of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED (VLSI) SYSTEMS, vol. 6, pp. 186–187, June 1998.



(a) (b)

Fig. 1. Decomposition of output $r$ of `alu4` (a) without and (b) with support minimization.

functions during disjoint functional decomposition subsumes nondisjoint decomposition as a special case.

Fig. 1 shows an output of benchmark circuit `alu4` when decomposed a) without regard to the support of the subfunctions and b) with subfunctions having minimal support. Instead of five eight-input subfunctions, the decomposition with support minimization yields two eight-input subfunctions, one five-input subfunction and two one-input "subfunctions." The subfunctions with one input actually cause a nondisjoint decomposition with the common variables $j$ and $k$.

Recently, Huang *et al.* presented a decomposition algorithm that finds subfunctions independent of certain bound variables [10]. Since the algorithm assigns just one code to each compatible class (called "strict" decomposition), not all functions independent of certain bound variables can be computed. Cong and Hwang also presented an algorithm to compute support minimized subfunctions [11]. This algorithm explicitly checks whether subfunctions exist that depend only on a certain subset of the bound variables. Due to its large complexity this algorithm is only efficient for small bound sets.

This article presents a new algorithm that computes subfunctions $d_i$ with minimal support. The algorithm iteratively computes support-minimal subfunctions. In each iteration step, sets of subfunctions are represented and computed implicitly using characteristic functions, which are represented by BDD's. In contrast to known methods [10], we do not assign the same code to all the vertices of a compatible class. By dealing with classes of bound set vertices we can keep the average complexity of the algorithm low such that large bound sets can be handled. Using different encodings for compatible bound set vertices ("nonstrict" decomposition) as well as being able to handle large bound sets provides a more general solution of the problem and improves the result quality. We will also show that computing support minimized subfunctions can be easily combined with an efficient approach for the decomposition of multiple-output functions [7].

## II. PRELIMINARIES

A *single-output Boolean function* of $b$ variables is given by $f : \{0,1\}^b \rightarrow \{0,1\}$. A *multiple-output Boolean function* is a vector of single-output Boolean functions and is denoted by a bold letter, $\mathbf{f} = (f_1, \cdots, f_m)$. Vectors of *Boolean variables* $x_i$ are also printed bold, $\mathbf{x} = (x_1, \cdots, x_b)$. The *support* of a function is the set of variables it depends on.

A *partition* $\Pi$ of the set $\mathcal{X}$ divides the set into disjoint *blocks* or *classes*. The *rank* of the partition, $|\Pi|$, is the number of blocks it contains. Let $R$ be an equivalence relation on $\mathcal{X}$. Then, the set of equivalence classes under $R$ is a partition of $\mathcal{X}$, denoted by $\mathcal{X}/R = \{C_1, \cdots, C_{|\mathcal{X}/R|}\}$. Let $\Pi_1 = \mathcal{X}/R_1$ and $\Pi_2 = \mathcal{X}/R_2$ be partitions of $\mathcal{X}$. Then $\Pi_2$ *refines* $\Pi_1$ if every block of $\Pi_2$ is contained in a block of $\Pi_1$. The *product* partition $\Pi_{\text{prod}}$ of $\Pi_1$ and $\Pi_2$, denoted $\Pi_{\text{prod}} = \Pi_1 \cdot \Pi_2$, is the smallest partition of $\mathcal{X}$ (i.e., with the smallest number of blocks) which refines both $\Pi_1$ and $\Pi_2$. The product of $c$ partitions $\Pi_i$ is $\Pi_{\text{prod}} = \prod_{i=1}^{c} \Pi_i$.

## III. BACKGROUND

### A. Classical Functional Decomposition

This section summarizes the classical functional decomposition theory [13], [14]. In particular, the notions *compatible* and *assignable* will be used in other sections.

Given a function $f$ and a partition of its $n$ input variables into the bound set BS $= \{x_1, \cdots, x_b\}$ and the free set FS $= \{y_1, \cdots, y_{n-b}\}$, functional decomposition determines *subfunctions* $d_1, \cdots, d_c$ and the *composition function* $g$ such that

$$\begin{aligned} &f(x_1, \cdots, x_b, y_1, \cdots, y_{n-b}) \\ &\quad = g(d_1(x_1, \cdots, x_b), \cdots, d_c(x_1, \cdots, x_b), y_1, \cdots, y_{n-b}). \quad (1) \end{aligned}$$

For nontrivial decompositions, we have $c < b$. Let $\mathcal{X} = \{0,1\}^b$ denote the set of BS-vertices. To check if a decomposition exists, we use the compatibility relation.

*Definition 1:* Two bound set vertices $\mathbf{x}_v$ and $\mathbf{x}_w$ are *compatible*, denoted $\mathbf{x}_v R_f \mathbf{x}_w$, if

$$\forall \mathbf{y} \in \{0,1\}^{n-b} : f(\mathbf{x}_v, \mathbf{y}) = f(\mathbf{x}_w, \mathbf{y}).$$

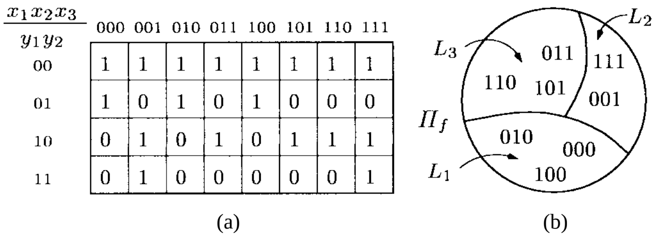
For completely specified functions, compatibility is an equivalence relation. The relation $R_f$ induces a *compatibility partition* $\Pi_f = \mathcal{X}/R_f = \{L_1, \cdots, L_\ell\}$ of the BS-vertices $\mathbf{x}$ into $\ell$ *compatible classes*. In the *decomposition chart* [14], compatibility of BS-vertices is visualized by identical columns. The column multiplicity in the decomposition chart equals the number $\ell$ of compatible classes.

The *decomposition condition* states that a decomposition according to (1) using subfunction vector $\mathbf{d}$ exists if and only if

$$\forall \mathbf{x}_v, \mathbf{x}_w \in \mathcal{X} : \neg(\mathbf{x}_v R_f \mathbf{x}_w) \Rightarrow \mathbf{d}(\mathbf{x}_v) \neq \mathbf{d}(\mathbf{x}_w). \quad (2)$$

We choose the minimum number of subfunctions $c = \lceil \mathrm{ld} \ell \rceil$. The decomposition condition says that different *codes* $\mathbf{d}(\mathbf{x})$ are required for incompatible BS-vertices, but either identical or different codes are allowed for compatible BS-vertices. Using different codes for compatible vertices is allowed if not all codes must be employed, i.e., if $\ell < 2^c$.

*Coding the bound set vertices is equivalent to determining the subfunction vector* $\mathbf{d}$. A common way to determine subfunctions is to assign a unique code to each compatible class [3], [10]. Such a decomposition is called *strict.* We adopt a different, *nonstrict* decomposition procedure. Our procedure selects subfunctions iteratively. Thus, the codes of all compatible classes are determined concurrently bit by bit. In each step of such a procedure, many functions $h : \{0,1\}^b \rightarrow \{0,1\}$ are suitable as a subfunction. These functions are called



| $x_1x_2x_3$ / $y_1y_2$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 01 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 10 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

Fig. 2. (a) Decomposition chart and (b) compatibility partition.

TABLE I
SOME FUNCTIONS $h : \{0,1\}^3 \rightarrow \{0,1\}$

| BS-vertices<br>(000) $\in L_1$ | (001) $\in L_2$ | (010) $\in L_1$ | (011) $\in L_3$ | (100) $\in L_1$ | (101) $\in L_3$ | (110) $\in L_3$ | (111) $\in L_2$ | function $h$ | assign. |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $\mathbf{0}$ | no |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | $x_1x_2x_3$ | no |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | $x_1x_2\overline{x_3}$ | no |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | $x_1x_2$ | no |
|  |  |  |  | $\vdots$ |  |  |  | $\vdots$ | $\vdots$ |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | $\overline{x_2}x_3 + \overline{x_1}x_3$ | no |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | $x_3$ | yes |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | $\overline{x_2}x_3 + \overline{x_1}x_3$ $+x_1x_2\overline{x_3}$ | yes |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | $x_1x_2 + x_3$ | yes |
|  |  |  |  | $\vdots$ |  |  |  | $\vdots$ | $\vdots$ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $\mathbf{1}$ | no |

*assignable*. A detailed explanation of this term is given in [7], [15]. For ease of explanation we only consider the first iteration step in the sequel.

*Property 1:* In this first iteration step of the iterative procedure, a function $h$ is *assignable* if and only if neither onset nor offset contain vertices of more than $2^{c-1}$ compatible classes.

*Example 1:* Let us illustrate the detection of assignable subfunctions in the first iteration step. Fig. 2(a) shows the decomposition chart for function $f(x_1, x_2, x_3, y_1, y_2) = \overline{x_1}\,\overline{x_3}\,\overline{y_1} + x_3\,\overline{y_2} + \overline{x_1}\,\overline{x_2}\,x_3y_1 + \overline{x_2}\,\overline{x_3}\,\overline{y_1} + x_1x_2\overline{y_2} + x_1x_2x_3y_1$. Fig. 2(b) shows the partition $\Pi_f = \{L_1, L_2, L_3\}$ of the set of BS-vertices $\mathcal{X} = \{0,1\}^3$. For example, BS-vertices (001) and (111) are compatible because the second and the last column in the decomposition chart are identical. They build the compatible class $L_2$.

Table I shows some functions $h : \{0,1\}^3 \rightarrow \{0,1\}$ and indicates their assignability in the first step of the iterative procedure decomposing $f$ of Fig. 2(a). The function $d(\mathbf{x}) = x_3$, which causes a nondisjoint decomposition, is also assignable because vertices of not more than two classes are contained in the onset (classes $L_2$ and $L_3$) and in the offset (classes $L_1$ and $L_3$). Since class $L_3$ overlaps both on- and offset, we have a nonstrict decomposition. This subfunction cannot be found by a strict decomposition technique where complete compatible classes are coded with a unique code.

### B. A General Implicit Decomposition Algorithm

We now describe a general implicit algorithm for single-output decomposition. This algorithm, which has been used in an extended version in [7], is the basis for the new algorithm presented in Section IV.

TABLE II
THE ONSET OF $\chi(\mathbf{e})$ REPRESENTING ALL ASSIGNABLE SUBFUNCTIONS

| $L_1$<br>(000)<br>$e_0$ | (010)<br>$e_2$ | (100)<br>$e_4$ | $L_2$<br>(001)<br>$e_1$ | (111)<br>$e_7$ | $L_3$<br>(011)<br>$e_3$ | (101)<br>$e_5$ | (110)<br>$e_6$ |
|---|---|---|---|---|---|---|---|
| – | – | – | 0 | 0 | 1 | 1 | 1 |
| – | – | – | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | – | – | 1 | 1 | 1 |
| 1 | 1 | 1 | – | – | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | – | – | – |
| 1 | 1 | 1 | 0 | 0 | – | – | – |

Let $h : \{0,1\}^b \rightarrow \{0,1\}$ be a function of the bound variables. To implicitly represent $h$-functions, we employ a bijective mapping from the set of all functions of the bound variables, which has cardinality $2^{2^b}$ to $\{0,1\}^p$ where $p = |\mathcal{X}| = 2^b$. A tuple $\mathbf{e} = (e_0, \cdots, e_{p-1}) \in \{0,1\}^p$ then represents a function $h(\mathbf{x})$. The variable $e_j$ assumes value "1" if the vertex $(j)$ is contained in the onset of $h$, and it assumes value "0" if the vertex $(j)$ is contained in the offset of $h$. For example, the function $d = x_3$ is represented by the minterm $\mathbf{e} = (01010101)$. A set of $h$-functions is represented by the onset of a characteristic function and thus in a single BDD.

Using Property 1, we compute the characteristic function $\chi(\mathbf{e})$ that implicitly represents the set of all assignable subfunctions for the first iteration step. Table II shows the onset of function $\chi(\mathbf{e})$ for the example of Fig. 2(a). After selecting a subfunction from the set of assignable functions in the first step, a new $\chi(\mathbf{e})$ is computed for the next iteration step. A more detailed description of the iterative decomposition algorithm can be found in [7].

## IV. NEW ALGORITHM

In order to compute subfunctions with minimal support, three problems have to be solved. First, how do we compute subfunctions that are independent of a single bound variable $x_i$? Second, how can this be done efficiently? Third, how are subfunctions calculated that are independent of a maximal number of bound variables?

### A. Computing Subfunctions Independent of a Single Variable $x_i$

We first show how to implicitly compute subfunctions that are independent of a certain bound variable $x_i$. For discussion, let us consider two BS-vertices that differ only in the value of $x_i$. Such vertices are called *$x_i$-adjacent*. An *$x_i$-adjacency pair* is a pair of $x_i$-adjacent vertices.

If the onset of a Boolean function $h(\mathbf{x})$ contains just one vertex of an $x_i$-adjacency pair, the onset representation must depend on $x_i$. In order to obtain a function independent of $x_i$, we have to assure that each adjacency pair is completely contained in either the onset or the offset. Based on this condition, we state a formula to compute all these functions implicitly,

$$\mathcal{I}_{x_i}(\mathbf{e}) = \prod_{j=0}^{2^b-1} \left(e_j\hat{e}_j + \overline{e_j}\,\overline{\hat{e}_j}\right) \qquad (3)$$

where $b$ is the number of bound variables, $e_j$ is the variable representing the BS-vertex $(j)$, and $\hat{e}_j$ is the variable representing its adjacent BS-vertex. Let us interpret formula (3). The term $(e_j\hat{e}_j + \overline{e_j}\,\overline{\hat{e}_j})$ is 1 if the adjacency pair that contains the BS-vertex $(j)$ either belongs to the onset $(e_j\hat{e}_j)$ or the offset $(\overline{e_j}\,\overline{\hat{e}_j})$ of a function $h(\mathbf{x})$. $\mathcal{I}_{x_i}(\mathbf{e})$ is 1 for a given tuple $\mathbf{e}$ if this condition holds for each BS-vertex, i.e., $\mathbf{e}$ then represents an $x_i$-independent function.

Now, we can easily calculate all assignable subfunctions that are independent of $x_i$. The set of assignable functions is represented by $\chi(\mathbf{e})$ and the set of $x_i$-independent functions by $\mathcal{I}_{x_i}(\mathbf{e})$. Therefore, we implicitly compute the set of assignable functions that are independent of $x_i$ as the product $\chi(\mathbf{e}) \cdot \mathcal{I}_{x_i}(\mathbf{e})$.

*Example 2:* Let us compute all subfunctions independent of $x_1$ for the function $f$ of Example 1. The $x_1$-adjacency pairs are $\langle(000),(100)\rangle$, $\langle(010),(110)\rangle$, $\langle(001),(101)\rangle$, and $\langle(011),(111)\rangle$. We implicitly represent all $x_1$-independent subfunctions using formula (3)

$$\begin{aligned}\mathcal{I}_{x_1}(\mathbf{e}) = {} & (e_0e_4 + \overline{e_0}\,\overline{e_4})(e_2e_6 + \overline{e_2}\,\overline{e_6})\\ & \cdot (e_1e_5 + \overline{e_1}\,\overline{e_5})(e_3e_7 + \overline{e_3}\,\overline{e_7}).\end{aligned}$$

According to Table II, the set of all assignable functions is given by

$$\begin{aligned}\chi(\mathbf{e}) = {} & \overline{e_1}e_3e_5e_6\overline{e_7} + e_1\overline{e_3}\,\overline{e_5}\,\overline{e_6}e_7 + \overline{e_0}\,\overline{e_2}e_3\overline{e_4}e_5e_6\\ & + e_0e_2\overline{e_3}e_4\overline{e_5}\,\overline{e_6} + \overline{e_0}\,e_1\overline{e_2}\,\overline{e_4}e_7 + e_0\overline{e_1}e_2e_4\overline{e_7}.\end{aligned}$$

Computing the set of assignable subfunctions independent of $x_1$ yields

$$\chi(\mathbf{e}) \cdot \mathcal{I}_{x_1}(\mathbf{e}) = \overline{e_0}e_1\overline{e_2}e_3\overline{e_4}e_5\overline{e_6}e_7 + e_0\overline{e_1}e_2\overline{e_3}e_4\overline{e_5}e_6\overline{e_7}.$$

Minterm $\overline{e_0}e_1\overline{e_2}e_3\overline{e_4}e_5\overline{e_6}e_7$ represents the function $d = x_3$, minterm $e_0\overline{e_1}e_2\overline{e_3}e_4\overline{e_5}e_6\overline{e_7}$ the function $d = \overline{x_3}$. Thus, there are two subfunctions independent of $x_1$.

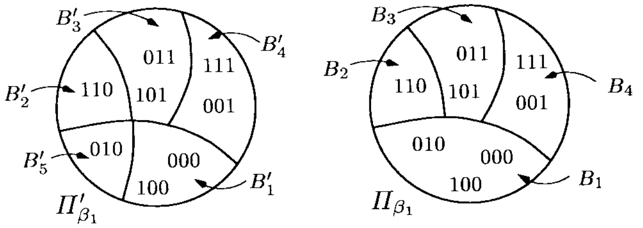
### *B. Increasing the Efficiency by Partitioning*

Up to now, we explained how to compute all subfunctions independent of $x_i$ working with BS-vertices. As the number of BS-vertices grows exponentially with the number of bound variables, the number of variables $e_j$, which $\chi(\mathbf{e})$ and $\mathcal{I}_{x_i}(\mathbf{e})$ depend on, becomes large even for small bound sets. This would yield large BDD's and limit the algorithm's efficiency. The problem is how to achieve efficiency for larger bound sets.

We suggest to group BS-vertices into classes, and associate variables $e_j$ with classes instead of individual vertices. These classes form a partition $\Pi$ of the set of BS-vertices. We use the classes of $\Pi$ to build functions $h(\mathbf{x})$ by assigning each class to either the on- or offset of $h(\mathbf{x})$. Such a function $h(\mathbf{x})$ is called *constructable* with respect to $\Pi$ [7].

By choosing a certain partition $\Pi$, we obtain a tradeoff between the efficiency of our algorithm and the quality of its results. The efficiency of our algorithm increases with a decreasing number of variables $e_j$ and thus a decreasing number of classes of $\Pi$. The result quality increases with an increasing number of functions constructable with respect to $\Pi$ and thus an increasing number of classes of $\Pi$.

Note that we may only choose a partition $\Pi$ that equals $\Pi_f$ or refines $\Pi_f$. At first sight, the compatibility partition $\Pi_f$ seems to be a reasonable choice of $\Pi$ as it has a small number of classes. However, it can be shown that we may not be able to find assignable functions independent of $x_i$ if we compute $x_i$-independent functions based on $\Pi_f$ [16].



(a) (b)

Fig. 3. Partition (a) $\Pi'_{\beta_1}$ and (b) $\Pi_{\beta_1}$.

In order to compute subfunctions independent of $x_i$ efficiently, we have to solve the following problem. Determine a partition $\Pi_{\beta_i}$ such that

- $\Pi_{\beta_i}$ refines $\Pi_f$,
- if there exists any assignable function independent of $x_i$, then at least one is constructable with respect to $\Pi_{\beta_i}$, and
- $\Pi_{\beta_i}$ comprises a minimum number of classes.

We state a two step algorithm to solve this problem:

*Step 1:* Merge BS-vertices into one class if they are compatible and their $x_i$-adjacent vertices are also compatible. The obtained partition is called $\Pi'_{\beta_i} = \{B'_1, \cdots\}$.

*Step 2:* Merge a class $B'_j \in \Pi'_{\beta_i}$ that contains only adjacency pairs with any other class $B'_k \in \Pi'_{\beta_i}$ that contains vertices compatible with the vertices of $B'_j$.

The obtained partition of the set of BS-vertices is called the *basis partition* $\Pi_{\beta_i}$.

*Example 3:* Let us perform Step 1 of our algorithm and compute partition $\Pi'_{\beta_1}$ for function $f$ of Fig. 2(a). Vertices (011) and (101) are grouped into one class as they are compatible and their adjacent vertices (111) and (001) are also compatible. Vertex (110) cannot be grouped with any other vertex as it is the only vertex of the compatible class $L_3$ that is adjacent to a vertex of compatible class $L_1$. Thus, we obtain $\Pi'_{\beta_1}$ as shown in Fig. 3(a).

Now, we perform Step 2. Class $B'_1 \in \Pi'_{\beta_1}$ is made up of the adjacency pair $\langle(000),(100)\rangle$. As these vertices are compatible with vertex (010) of class $B'_5$, the classes $B'_1$ and $B'_5$ are merged. We obtain the basis partition $\Pi_{\beta_1}$ as shown in Fig. 3(b).

Please note an important property of $\Pi_{\beta_i}$. The vertices of each class have their adjacent vertices in at most one other class. Therefore, only pairs of classes of $\Pi_{\beta_i}$ have to be commonly contained in the on- or offset of a function independent of $x_i$.

We can now compute all subfunctions that are independent of $x_i$ and constructable based on $\Pi_{\beta_i}$. Similar to (3), this is done implicitly using

$$\mathcal{I}_{x_i}(\mathbf{e}) = \prod_{j=1}^{|\Pi_{\beta_i}|} \left(e_j\hat{e}_j + \overline{e_j}\,\overline{\hat{e}_j}\right) \tag{4}$$

where $e_j$ now represents a class $B_j \in \Pi_{\beta_i}$ and not a vertex as before; $\hat{e}_j$ represents the class $B_k \in \Pi_{\beta_i}$ "adjacent" to $B_j$,

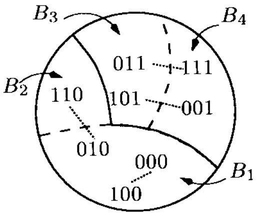


Fig. 4. Partition $\Pi_{\beta_1}$ with $x_1$-adjacency pairs.

i.e., the class containing vertices that are adjacent to vertices of $B_j$. Then, the characteristic function $\mathcal{P}_{x_i}(\mathbf{e})$ of all assignable functions that are independent of $x_i$ and constructable with respect to $\Pi_{\beta_i}$ is

$$\mathcal{P}_{x_i}(\mathbf{e}) = \chi(\mathbf{e}) \cdot \mathcal{I}_{x_i}(\mathbf{e}) \tag{5}$$

where $\chi(\mathbf{e})$ represents the set of assignable functions that are constructable with respect to $\Pi_{\beta_i}$. These subfunctions are called *s-preferable* (support-preferable) functions with respect to $x_i$.

The following theorem gives the condition on which resorting to s-preferable functions is sufficient to obtain subfunctions independent of variable $x_i$.

*Theorem 1:* In the first step of the iterative decomposition algorithm, there exists a function s-preferable with respect to $x_i$ if and only if there exists an assignable function independent of $x_i$.

The proof of Theorem 1 can be found in the Appendix.

It can also be shown that the partition $\Pi_{\beta_i}$ is the smallest partition such that Theorem 1 holds. Experimental results show that the set of s-preferable functions is much smaller than the set of all assignable functions independent of $x_i$.

*Example 4:* We compute the set of s-preferable functions. Now, we have a variable $e_j$ for each class $B_j \in \Pi_{\beta_1}$. First, we determine $\chi(\mathbf{e})$, i.e., the set of assignable subfunctions constructable w.r.t. $\Pi_{\beta_1}$

$$\chi(\mathbf{e}) = e_2e_3\overline{e_4} + \overline{e_2}\,\overline{e_3}e_4 + \overline{e_1}e_2e_3 + e_1\overline{e_2}\,\overline{e_3} + \overline{e_1}e_4 + e_1\overline{e_4}.$$

To compute $\mathcal{I}_{x_1}(\mathbf{e})$, which represents $x_1$-independent functions, we determine pairs of "adjacent" classes. As can be seen in Fig. 4, classes $B_1$ and $B_2$ as well as classes $B_3$ and $B_4$ are such pairs.

Therefore, we have

$$\mathcal{I}_{x_1}(\mathbf{e}) = (e_1e_2 + \overline{e_1}\,\overline{e_2})(e_3e_4 + \overline{e_3}\,\overline{e_4})$$

.

Now, we compute the set of s-preferable functions

$$\mathcal{P}_{x_1}(\mathbf{e}) = \chi(\mathbf{e}) \cdot \mathcal{I}_{x_1}(\mathbf{e}) = \overline{e_1}\,\overline{e_2}e_3e_4 + e_1e_2\overline{e_3}\,\overline{e_4}.$$

There are two s-preferable functions: $d(\mathbf{x}) = x_3$ represented by $\overline{e_1}\,\overline{e_2}e_3e_4$ and $d(\mathbf{x}) = \overline{x_3}$ represented by $e_1e_2\overline{e_3}\,\overline{e_4}$.

We have shown how to find subfunctions independent of a certain bound variable $x_i$ efficiently. We have not addressed how to find subfunctions independent of several bound variables. This problem is solved in the next section.

### C. Computing Subfunctions Independent of Several Variables

So far, we computed a basis partition $\Pi_{\beta_i}$ for a *single* bound variable $x_i$. In order to compute s-preferable functions that are independent of several bound variables, we introduce the basis partition $\Pi_\beta$ for *all* bound variables. The *basis partition* $\Pi_\beta$ is defined as

$$\Pi_\beta = \prod_{i \text{ with } \mathcal{P}_{x_i} \neq \mathbf{0}} \Pi_{\beta_i}. \tag{6}$$

To keep the number of classes of $\Pi_\beta$ small, the product is only computed over partitions $\Pi_{\beta_i}$ of those variables $x_i$ for which s-preferable functions exist at all, i.e., $\mathcal{P}_{x_i} \neq \mathbf{0}$. The following theorem expresses the meaning of the basis partition $\Pi_\beta$:

*Theorem 2:* The basis partition $\Pi_\beta$ is the partition with the smallest number of classes such that all subfunctions s-preferable with respect to any single bound variable $x_i$ can be computed.

The proof of Theorem 2 can be found in the Appendix.

After we have computed $\mathcal{P}_{x_i}(\mathbf{e})$ based on $\Pi_\beta$ for each bound variable, we choose a minterm $\mathbf{e}$ such that $\mathcal{P}_{x_i}(\mathbf{e}) = 1$ for a maximum number of $x_i$. The subfunction represented by this minterm $\mathbf{e}$ then depends on a minimal number of bound variables. In order to find this minterm $\mathbf{e}$, we build a matrix where each column corresponds with a minterm $\mathbf{e}$ and each row corresponds with a function $\mathcal{P}_{x_i}$. An entry for column $\mathbf{e}$ and row $\mathcal{P}_{x_i} = 1$ if $\mathcal{P}_{x_i}(\mathbf{e}) = 1$. Selecting a subfunction with minimal support then corresponds with selecting a column with the maximum number of "1"'s. Since this problem is equivalent to selecting an optimal subfunction during multiple-output decomposition, we use the *maxcol* algorithm of [15]. This algorithm, which represents the matrix by a single BDD, is similar to the *Lmax* algorithm that was suggested by Kam *et al.* [17]. A detailed description of the algorithm can be found in [15].

In order to compute subfunctions with minimal support in each iteration step of the decomposition algorithm, $\mathcal{P}_{x_i}(\mathbf{e})$ must be updated in each step. This is done using Formula (5) where $\chi(\mathbf{e})$ represents the functions that are assignable in the current iteration step. To conclude this section, we outline how to compute support-minimal subfunctions during multiple-output decomposition.

### D. Detecting S-Preferable Multiple-Output Decompositions

An approach to detect subfunctions which are concurrently assignable for several outputs is proposed in [7]. Extracting subfunctions that can be shared among several outputs reduces the circuit area. It is shown in [7] that during multiple-output decomposition of the function vector $\mathbf{f} = (f_1, \cdots, f_m)$ an optimum multiple-output decomposition can be obtained by considering only assignable subfunctions which are constructable with respect to $\hat{\Pi}$

$$\hat{\Pi} = \prod_{k=1}^{m} \Pi_{f_k}$$

where the compatibility partitions of the individual outputs are given by $\Pi_{f_k}$. Shared subfunctions are then found by solving

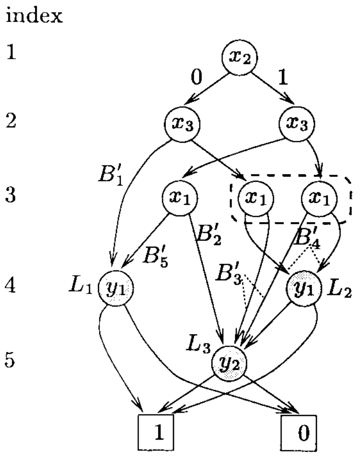


Fig. 5. BDD for example function $f$.

a covering problem similar to the one described in the last section.

Our goal now is to detect $x_i$-independent subfunctions among the set of shared subfunctions. Quite similar to the previous discussion about how to compute s-preferable subfunctions that are independet of several bound variables, we need a common basis to represent the set of shared subfunctions as well as s-preferable functions. These shared subfunctions that are independent of several bound variables can be computed using the partition $\hat{\Pi}_\beta$

$$\hat{\Pi}_\beta = \hat{\Pi} \cdot \prod_{k=1}^{m} \Pi_\beta^{f_k} \tag{7}$$

where the basis partition of an individual output is given by $\Pi_\beta^{f_k}$. The multiplication with $\hat{\Pi}$ is necessary in order to be able to compute all functions that are constructable with respect to $\hat{\Pi}$ such that an optimum multiple-output decomposition can be obtained.

## V. IMPLICIT COMPUTATION OF PARTITION $\Pi_{\beta_i}$

In order to compute the basis partition $\Pi_{\beta_i}$, we use a BDD of function $f$, where bound variables must be ordered before free variables, and $x_i$ is the bound variable directly before the free variables. The BDD for the function $f$ of the continued example is shown in Fig. 5. By *cut_nodes*$(b)$, drawn grey in Fig. 5, we denote the set of BDD nodes which have an *index* $> b$ and at least one predecessor with *index* $\leq b$. All paths going to one node $\nu \in$ *cut_nodes*$(b)$ correspond with vertices of a compatibility class $L \in \Pi_f$ [3]. In Fig. 5 where $b = 3$, there are three compatibility classes $L_1$, $L_2$, and $L_3$. The predecessors of the nodes $\nu \in$ *cut_nodes*$(b)$ form a set of nodes, denoted *preset*$(b)$. A pair of nodes $\nu_1, \nu_2 \in$ *preset*$(b)$ is now merged to a meta-node if they have the same successors. In our example, the pair of nodes circled by the dashed line is merged to a meta-node.

Now, those paths containing an edge (pair) from a (meta) node $\in \mathit{preset}(b)$ with $\mathit{index} = b$ to a node $\in \mathit{cut\_nodes}(b)$ represent a class $B'_j$. All paths that contain an edge from a node $\in \mathit{preset}(b)$ with $\mathit{index} \leq b - 1$ to a node $\in \mathit{cut\_nodes}(b)$ represent a class $B'_j \subseteq L_k$ which contains only adjacency pairs. Such a class must be unified with another class $B'_i \subseteq L_k$. In the example, class $B'_1$ is unified with $B'_5$. After taking all such unions, the classes $B_j \in \Pi_{\beta_i}$ have been computed.

TABLE III
PROBLEM CHARACTERISTICS

| $f$ | sup. of $f$ | $b$ | $\ell$ | $\lvert\Pi_\beta\rvert$ | #indep. | #assign. | #s-pref. | sup. of | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | $d_1$ | $d_2$ | $d_3$ |
| $f_{clip}$ | 9 | 5 | 6 | 20 | 65536 | 628138 | 96 | 1 | 3 | 5 |
| $f_{ex2}$ | 16 | 7 | 3 | 21 | $1.8 \cdot 10^{19}$ | $3.0 \cdot 10^{20}$ | 16 | 5 | 7 | - |
| $f_{ind4}$ | 104 | 53 | 7 | 38 | $\approx 10^{10^{15}}$ | $> 10^{308}$ | $2.7 \cdot 10^6$ | 8 | 16 | 53 |

Note that $x_i$-adjacent BS-vertices are represented by paths passing through the same node $\in \mathit{preset}(b)$. So, we can determine adjacent classes by simply evaluating the predecessor relations in the BDD while we compute $\Pi_{\beta_i}$. Therefore, we get all information we need to compute $\Pi_{\beta_i}$ and $\mathcal{I}_{x_i}$ by a reordering of the BDD and a subsequent BDD traversal.

## VI. EXPERIMENTAL RESULTS

The implicit algorithm for single-output decomposition was implemented in the program *ISODEC-S* (implicit single-output decomposition with support minimization), which is embedded into the synthesis tool TOS–TUM.[2]

The experimental data in Table III give some typical problem parameters. The data were gathered during the decomposition of single-output functions in the benchmark circuits `clip,example2` and the industrial benchmark `ind4`. The support size of $f$, the number of bound variables, the number of compatible classes $\ell$, and the number of classes of $\Pi_\beta$, which is the number of levels of BDD's representing characteristic functions, are given in columns 2–5. The number of functions that are independent of a certain variable (*#indep.*) and assignable in the first iteration step (*#assign.*) are shown next. The maximum number of functions s-preferable with respect to a single variable $x_i$ is given in column 8. The support sizes of the extracted subfunctions are shown next.

The number of classes of $\Pi_\beta$ is typically small compared with the number of BS-vertices, $2^b$. This translates into a set of s-preferable functions which is also small compared with the other sets. The reduction of inputs of the extracted subfunctions is apparent. As one subfunction of $f_{\text{clip}}$ depends on only one input, a nondisjoint decomposition has been performed. For $f_{\text{ind4}}$, the first selected subfunction depends on only 8 out of 53 bound variables.

### *A. Reductions in LUT Count*

Table IV shows the effectiveness of the new single-output decomposition approach in reducing the LUT count if two-level networks are decomposed (multilevel circuits were collapsed before decomposition). We applied our decomposition algorithm recursively to obtain functions with at most 5 inputs. A variable partitioning heuristic similar to the heuristic presented in [18] was used here targeting minimal LUT count.

The number of primary inputs ($\#I$) and outputs ($\#O$) are given in column 2 and 3. Columns 4, 5, and 6, show the LUT count, the circuit depth, and CPU time (DEC AlphaStation 250

---

[2] TOS has been developed at the Technical University of Munich, Germany.

TABLE IV
DECOMPOSITION OF TWO-LEVEL CIRCUITS

| | | | usual | | | **ISODEC–S** | | |
|---|---|---|---|---|---|---|---|---|
| net | #I | #O | #LUT | depth | CPU | **#LUT** | **depth** | **CPU** |
| 9sym | 9 | 1 | 8 | 3 | 0.1 | 8 | 3 | 0.4 |
| alu2 | 10 | 6 | 55 | 6 | 1.1 | 52 | 6 | 7.3 |
| alu4 | 14 | 8 | 316 | 11 | 9.6 | 231 | 11 | 210.3 |
| apex2 | 39 | 3 | 328 | 16 | 113.3 | 165 | 11 | 161.9 |
| apex6 | 135 | 99 | 213 | 6 | 8.5 | 207 | 6 | 8.3 |
| apex7 | 49 | 37 | 76 | 5 | 7.5 | 71 | 5 | 7.9 |
| b9 | 41 | 21 | 57 | 5 | 0.6 | 41 | 4 | 0.9 |
| clip | 9 | 5 | 22 | 3 | 0.4 | 22 | 3 | 0.5 |
| duke2 | 22 | 29 | 258 | 8 | 6.3 | 213 | 8 | 21.5 |
| example2 | 85 | 66 | 156 | 5 | 2.2 | 139 | 4 | 2.4 |
| frg1 | 28 | 3 | 116 | 11 | 12.1 | 40 | 9 | 55.7 |
| i7 | 199 | 67 | 139 | 2 | 0.3 | 103 | 2 | 0.6 |
| misex2 | 25 | 18 | 41 | 4 | 0.4 | 40 | 4 | 0.4 |
| misex3c | 14 | 14 | 200 | 10 | 4.3 | 152 | 8 | 29.1 |
| rd84 | 8 | 4 | 12 | 2 | 0.2 | 12 | 2 | 0.5 |
| sao2 | 10 | 4 | 25 | 4 | 0.5 | 22 | 3 | 7.0 |
| too_large | 38 | 3 | 328 | 16 | 113.3 | 165 | 11 | 163.4 |
| vda | 17 | 39 | 495 | 7 | 10.5 | 351 | 8 | 54.4 |
| vg2 | 25 | 8 | 79 | 9 | 3.5 | 59 | 9 | 5.7 |
| $\sum$(MCNC) | 777 | 435 | 2924 | 133 | 294.7 | 2093 | 117 | 738.2 |
| perc. | - | - | 100 % | 100 % | - | **-28.4** % | **-12.0** % | - |
| ind1 | 36 | 9 | 220 | 6 | 39.4 | 185 | 6 | 41.9 |
| ind2 | 17 | 27 | 315 | 8 | 20.9 | 263 | 7 | 30.7 |
| ind3 | 33 | 27 | 1362 | 15 | 111.5 | 881 | 11 | 392.2 |
| ind4 | 20 | 27 | 7427 | 24 | 3325.2 | 2367 | 18 | 3352.4 |
| ind5 | 90 | 22 | 157 | 4 | 7.3 | 136 | 4 | 8.8 |
| ind6 | 237 | 218 | 341 | 7 | 29.0 | 322 | 7 | 29.3 |
| ind7 | 42 | 479 | 494 | 8 | 41.4 | 434 | 8 | 47.7 |
| ind8 | 22 | 42 | 1077 | 14 | 56.4 | 915 | 16 | 159.4 |
| $\sum$(IND) | 497 | 851 | 11393 | 86 | 3631.1 | 5503 | 77 | 4062.4 |
| perc. | - | - | 100 % | 100 % | - | **-51.7** % | **-10.5** % | - |

4/266), respectively, of single-output decomposition without support minimization (*usual*) where subfunctions are chosen randomly. The columns under *ISODEC-S* show the results of the new approach.

An average LUT count reduction of 28.4% for the MCNC benchmarks demonstrates the potential of decomposition with support minimization. The reduction is even more impressive for the industrial benchmarks (IND) due to the extremely good result for `ind4`. Although, we do not explicitly consider delay information during decomposition, the circuit depth is reduced by 12.0% for the set of MCNC benchmarks and by 10.5% for the set of industrial benchmarks. This reduction is due to the fact that we have to perfom fewer decompositions to get a network with five-input nodes if we select support minimized subfunctions. Besides the area reduction this has the additional effect that also the circuit depth may be reduced. The increase in CPU time is acceptable. Decompositions with up to 53 bound variables (`ind4`) are performed. The number of classes in the basis partition, $|\Pi_\beta|$, ranges from 9 (`apex6`) up to 50 (`apex2,too_large,frg1,vda,ind4`) in this experiment.

TABLE V
RESULTS AFTER PLACEMENT AND ROUTING

| | usual | | | ISODEC–S | | |
|---|---|---|---|---|---|---|
| net | #CLB | part | delay | #CLB | part | delay |
| 9sym | 7 | 3120A | 24.1 | 7 | 3120A | 25.8 |
| alu2 | 47 | 3120A | 64.0 | 44 | 3120A | 49.2 |
| alu4 | 273 | 3190A | 126.6 | 201 | 3164A | 99.4 |
| apex2 | 265 | 3190A | 153.5 | 131 | 3164A | 104.9 |
| apex6 | 174 | - | - | 168 | - | - |
| apex7 | 57 | 3142A | 48.5 | 51 | 3142A | 49.9 |
| b9 | 44 | 3120A | 41.1 | 30 | 3120A | 39.1 |
| clip | 17 | 3120A | 30.4 | 16 | 3120A | 29.4 |
| duke2 | 200 | 3190A | 119.7 | 161 | 3164A | 69.1 |
| example2 | 117 | 3195A | 66.1 | 102 | 3195A | 54.6 |
| frg1 | 98 | 3142A | 85.3 | 35 | 3120A | 60.7 |
| i7 | 102 | - | - | 84 | - | - |
| misex2 | 33 | 3120A | 34.7 | 31 | 3120A | 35.3 |
| misex3c | 165 | 3164A | 110.7 | 121 | 3142A | 72.8 |
| rd84 | 10 | 3120A | 25.4 | 10 | 3120A | 23.4 |
| sao2 | 24 | 3120A | 31.3 | 21 | 3120A | 32.0 |
| too_large | 265 | 3190A | 164.3 | 131 | 3164A | 104.1 |
| vda | 432 | 3195A | 100.3 | 285 | 3190A | 92.6 |
| vg2 | 65 | 3130A | 60.0 | 49 | 3120A | 69.7 |
| $\sum$ | 2395 | - | 1286.0 | 1678 | - | 1012.0 |
| perc. | 100 % | - | 100 % | -29.9 % | - | -21.3 % |

## B. Technology Mapping for SRAM-Cell-Based FPGA's

We target the Xilinx XC3000 architecture, which has SRAM cells with five inputs and two outputs. We mapped the decomposed MCNC benchmark circuits of Table IV to this Xilinx XC3000 architecture. We also placed and routed the mapped circuits on Xilinx XC3100A FPGA's (3120APC68-4, 3130APC44-4, 3142APG132-4, 3164APG132-4, 3190APC84-4, 3195APQ208-4) using the Xilinx `ppr` tool. We selected the smallest part type such that the utilization of SRAM cells was below 80% as recommended in [19]. The results are shown in Table V. The results for the single-output decomposition approach without support minimization and our new single-output decomposition approach are given in the columns

TABLE VI
MAPPING TO XILINX XC3000 CLBS

| net | Algo. 3 #CLB | FGMap #CLB | FGSyn #CLB | SIS-1.3 #CLB | **ISODEC–S** **#CLB** | **CPU** |
|---|---|---|---|---|---|---|
| 9sym | 7 | 7 | - | 7 | 7 | 0.4 |
| alu2 | 54 | 53 | 55 | 76 | 44 | 7.3 |
| alu4 | 180 | - | 56 | 43 | 48 | 9.9 |
| apex2 | - | - | 60 | 55 | 75 | 20.0 |
| apex4 | 393 | 356 | - | 372 | 329 | 15.9 |
| apex6 | 186 | - | 181 | 154 | 126 | 0.6 |
| apex7 | 51 | 47 | 43 | 47 | 40 | 0.1 |
| clip | 24 | 20 | 18 | 26 | 16 | 0.8 |
| des | 865 | - | - | 710 | 493 | 12.4 |
| duke2 | 105 | 178 | 85 | 109 | 121 | 7.0 |
| f51m | 12 | 11 | 8 | 11 | 9 | 0.03 |
| misex1 | 11 | 8 | 8 | 9 | 9 | 0.03 |
| misex2 | 28 | 21 | 22 | 21 | 21 | 0.04 |
| rd73 | 7 | 7 | 5 | 5 | 6 | 0.1 |
| rd84 | 12 | 12 | 8 | 10 | 10 | 0.5 |
| rot | 152 | 194 | 136 | 140 | 123 | 0.8 |
| sao2 | 32 | 27 | 25 | 28 | 21 | 7.0 |
| vg2 | 20 | 23 | 17 | 19 | 17 | 0.2 |
| example2 | - | - | - | 70 | 63 | 0.1 |
| vda | - | - | - | 173 | 139 | 1.4 |
| C499 | 62 | - | 54 | 66 | 50 | 0.1 |
| C880 | 84 | 74 | 87 | 76 | 74 | 0.7 |
| C1908 | - | - | 73 | 83 | 68 | 0.1 |
| C2670 | - | - | 122 | 113 | 113 | 0.5 |
| C5315 | - | - | 316 | 290 | 281 | 0.7 |
| C7552 | - | - | 317 | 275 | 265 | 2.0 |
| $\sum$(sub-Algo 3) | 2285 | - | - | - | 1564 | - |
| perc. | 100 % | - | - | - | **-31.6** % | - |
| $\sum$(sub-FGMap) | - | 1038 | - | - | 847 | - |
| perc. | - | 100 % | - | - | **-18.4** % | - |
| $\sum$(sub-FGSyn) | - | - | 1696 | - | 1537 | - |
| perc. | - | - | 100 % | - | **-9.4** % | - |
| $\sum$(all) | - | - | - | 2988 | 2568 | - |
| perc. | - | - | - | 100 % | **-14.1** % | - |

titled *usual* and *ISODEC-S*, respectively. The number of two-output SRAM cells (CLB's) are given in column 2 and 5. An abbreviation for the part type on which a certain circuit was implemented is shown in column 3 and 6. Column 4 and 7 show the worst case pad-to-pad delays of the circuits after placement and routing. These delays were computed with the Xilinx `xdelay` tool. Note that due to the large number of primary inputs and outputs, circuits `apex6` and `i7` cannot be implemented on a single XC3100A FPGA.

The average reduction in the number of CLB's is 29.9%. This reduction in the number of CLB's is even larger than the reduction of 28.4% in the number of LUT's. That shows that *ISODEC-S* not only generates a smaller number of nodes but also nodes with fewer inputs. Therefore, two nodes could more often be mapped into one two-output SRAM cell compared to the usual single-output decomposition approach. The reduced number of SRAM cells has also the advantage that in 8 out of 17 cases a smaller part type could be used to implement a circuit. All circuits that have been generated by *ISODEC-S* could be placed and routed without any problems. This shows that in contrast to [20] we do not have to sacrifice area in order to obtain routable designs. Even without considering the circuit delay during technology mapping, pad-to-pad delays were reduced by 21.3%. This improvement is achieved by the area and circuit depth reduction with *ISODEC-S*. It indicates that *ISODEC-S* significantly improves the mapping result.

Furthermore, we compared the results of our single-output decomposition algorithm with four state-of-the-art FPGA technology mapping approaches, which are *Algorithm 3* proposed by Huang *et al.* [10], *FGMap* proposed by Lai *et al.* [3], [21], *FGSyn* also proposed by Lai *et al.* [12], and *SIS-1.3* [22]. We have chosen *Algorithm 3* since it is a functional single-output decomposition method that also computes subfunctions with minimal support. *FGMap* has been chosen since it is a functional single-output decomposition method that performs nondisjoint decompositions as it is also done by our approach. FGSyn is, in contrast to our single-output decomposition approach, a functional multiple-output decomposition approach. It also performs nondisjoint decompositions. *SIS-1.3* combines various collapsing, decomposition and don't care optimization techniques.

As it was suggested in [21], large circuits have been preoptimized with *SIS* using the script *script.rugged* [22]. The results shown in column *Algo. 3* of Table VI are the best results reported for *Algorithm 3* in [10]. Results from [21] are repeated in column *FGMap*. Results for *FGSyn (bx-csn)* from [12] are repeated in colum *FGSyn*. Results for *SIS-1.3* shown in Table VI have been obtained using the FPGA synthesis script given in [22]. Column *ISODEC-S* gives the results of our new method. After technology mapping, the node functions were assigned to CLB's as permitted by the XC3000 technology.

*ISODEC-S* outperforms the single output decomposition approaches *Algorithm 3* and *FGMap* by 31.6% and 18.4%, respectively. It also outperforms the multiple-output decomposition approach *FGSyn* by 9.4%. The improvement when compared with *SIS-1.3* is 14.1%.

## VII. CONCLUSION

We analyzed the problem of support minimization during functional decomposition and proposed a new, implicit algorithm to compute subfunctions with minimal support. The efficiency of our algorithm mainly stems from the fact that we derived a suitable partitioning of the bound set vertices into classes and dealed with these classes instead of individual vertices. Therefore, it can handle large bound sets. Our algorithm is more general than the method of [10], since we perform nonstrict decompositions.

Experimental results show that the module count is reduced substantially and that the mapped circuits can be placed and routed without any problems. These results demonstrate the importance of the problem of support minimization during decomposition as well as the effectiveness of the new algorithm.

Currently, we are working on the problem of computing subfunctions with other properties like, e.g., symmetry. Combining this work with multiple-output decomposition and support minimization will lead to a more general understanding of the encoding problem in functional decomposition.

## APPENDIX

### A. Proof of Theorem 1

(Only if:) By definition any s-preferable function is assignable and independent of $x_i$.

(If:) We show how to build a subfunction $d$ which is s-preferable with respect to $x_i$ from an assignable function $d'$ which is $x_i$-independent but not s-preferable with respect to $x_i$. Since $d'$ is not constructable with respect to $\Pi_{\beta_i}$, there exist two vertices $\mathbf{x}_v$ and $\mathbf{x}_w$ such that these vertices are commonly contained in a class of $\Pi_{\beta_i}$, but $d'(\mathbf{x}_v) \neq d'(\mathbf{x}_w)$. As $\mathbf{x}_v$ and $\mathbf{x}_w$ are compatible, they might as well have identical code. There are two cases for $\hat{\mathbf{x}}_v$ which denotes the $x_i$-adjacent vertex to $\mathbf{x}_v$ and $\hat{\mathbf{x}}_w$ which denotes the $x_i$-adjacent vertex to $\mathbf{x}_w$: 1) If $\mathbf{x}_v$ and $\mathbf{x}_w$ are commonly contained in a class of $\Pi'_{\beta_i}$, $\hat{\mathbf{x}}_v$ and $\hat{\mathbf{x}}_w$ are compatible and can have identical code. We change function $d'$ such that $\mathbf{x}_w$ obtains the code associated with $\mathbf{x}_v$, and $\hat{\mathbf{x}}_w$ obtains the code associated with $\hat{\mathbf{x}}_v$. 2) If $\mathbf{x}_v$ and $\mathbf{x}_w$ are not commonly contained in a class of $\Pi'_{\beta_i}$, then $\mathbf{x}_v$ and $\mathbf{x}_w$ are commonly contained in a class of $\Pi_{\beta_i}$ due to Step 2 of the computation procedure of $\Pi_{\beta_i}$. Refering to the notation of Step 2, let $\mathbf{x}_v \in B'_k$ and $\mathbf{x}_w \in B'_j$. We change function $d'$ such that $\mathbf{x}_w$ and $\hat{\mathbf{x}}_w$, which are compatible, obtain the code associated with $\mathbf{x}_v$. The code of $\hat{\mathbf{x}}_v$ is not changed. This procedure can be repeated until a function $d$ which is s-preferable with respect to $x_i$ is obtained. □

### B. Proof of Theorem 2

By definition of the product of partitions, $\Pi_\beta$ is the partition with the smallest number of classes that refines all partitions $\Pi_{\beta_i}$ of the product of partitions. Since we omit each partition $\Pi_{\beta_i}$ of a bound variables $x_i$ in the product for which no s-preferable function exists at all, only a minimal refinement is done. Since $\Pi_\beta$ is a refinement of each considered $\Pi_{\beta_i}$, all functions that are constructable with respect to any considered $\Pi_{\beta_i}$ are also constructable with respect to $\Pi_\beta$. So, the set of functions which are s-preferable with respect to any $x_i$ is completely contained in the set of functions that are constructable with respect to $\Pi_\beta$. □

## ACKNOWLEDGMENT

The authors are very grateful to Prof. K. J. Antreich of the Technical University of Munich, Munich, Germany, for many valuable discussions and his steady interest in their work.

## REFERENCES

[1] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Logic synthesis for programmable gate arrays,” in *Proc. 27th ACM/IEEE Design Automation Conf., DAC,* June 1990, pp. 620–625.

[2] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Optimum functional decomposition using encoding,” in *Proc. 31st ACM/IEEE Design Automation Conf., DAC,* 1994, pp. 408–414.

[3] Y.-T. Lai, M. Pedram, and S. Vrudhula, “BDD based decomposition of logic functions with application to FPGA synthesis,” in *Proc. 30st ACM/IEEE Design Automation Conf., DAC,* June 1993, pp. 642–647.

[4] J. Cong and Y. Ding, “Beyond the combinatorial limit in depth minimization for LUT-based FPGA design,” in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD,* 1993.

[5] T. Sasao, *Logic Synthesis and Optimization*. New York: Kluwer Academic, 1993.

[6] T.-T. Hwang, R. M. Owens, M. J. Irwin, and K. H. Wang, “Logic synthesis for field-programmable gate arrays,” *IEEE Trans. Computer-Aided Design,* vol. 13, Oct. 1994.

[7] B. Wurth, K. Eckl, and K. Antreich, “Functional multiple-output decomposition: Theory and an implicit algorithm,” in *Proc. 32nd ACM/IEEE Design Automation Conf., DAC,* 1995, pp. 54–59.

[8] T. Stanion and C. Sechen, “A method for finding good Ashenhurst decompositions and its application to FPGA synthesis,” in *Proc. 32nd ACM/IEEE Design Automation Conf, DAC,* 1995, pp. 60–64.

[9] W. Z. Shen, J.-D. Huang, and S.-M. Chao, “Lambda set selection in Roth-Karp decomposition for LUT-based FPGA technology mapping,” in *Proc. 32nd ACM/IEEE Design Automation Conf., DAC,* 1995, pp. 65–69.

[10] J.-D. Huang, J.-Y. Jou, and W.-Z. Shen, “Compatible class encoding in Roth-Karp decomposition for two-output LUT architecture,” in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design ICCAD,* Nov. 1995, pp. 359–363.

[11] J. Cong and Y.-Y. Hwang, “Partially-dependent functional decomposition with applications in FPGA synthesis and mapping,” in *Fifth Int. Symp. Field-Programmable Gate Arrays,* Feb. 1997.

[12] Y.-T. Lai, K.-R. R. Pan, and M. Pedram, “OBDD—Based function decomposition: Algorithms and implementation,” *IEEE Trans. Computer-Aided Design,* vol. 15, pp. 977–990, Aug. 1996.

[13] J. P. Roth and R. M. Karp, “Minimization over Boolean graphs,” *IBM J.,* pp. 227–238, 1962.

[14] H. A. Curtis, *A New Approach to the Design of Switching Circuits.* Princeton, NJ: D. Van Nostrand, 1962.

[15] B. Wurth, K. Eckl, and K. Antreich, “Functional multiple-output decomposition for lookup-table based FPGA’s,” in *Proc. Workshop Notes Int. Workshop Logic Synth. IWLS,* CA, May 1995, pp. 9-26–9-37.

[16] C. Legl, B. Wurth, and K. Eckl, “An implicit algorithm for support minimization during functional decomposition,” in *Proc. European Design Test Conf. ED&TC,* Mar. 1996, pp. 412–417.

[17] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, “A fully implicit algorithm for exact state minimization,” in *Proc. 31st ACM/IEEE Design Automation Conf. DAC,* 1994, pp. 684–690.

[18] C. Legl, B. Wurth, and K. Eckl, “A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs,” in *Proc. 33rd ACM/IEEE Design Automation Conf., DAC,* June 1996, pp. 730–733.

[19] Xilinx Inc., *The Programmable Logic Data Book,* San Jose, CA, 1996.

[20] M. Schlag, J. Kong, and P. K. Chan, "Routability-driven technology mapping for lookup table-based FPGA's," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 13–26, Jan. 1994.

[21] Y.-T. Lai, K.-R. R. Pan, M. Pedram, and S. Sastry, "FGMap: A technology mapping algorithm for look-up table type FPGA's based on function graphs," in *Proc. Workshop Notes Int. Workshop Logic Synth. IWLS*, May 1993, pp. 9b1–9b4.

[22] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Electron. Res. Lab., Memo. UCB/ERL M92/41, pp. 1–45, May 1992.

**Christian Legl** received the Dipl.-Ing. degree in electrical and computer engineering from the Technical University of Munich, Munich, Germany, in 1994. He is currently working towards the Ph.D. degree in electrical and computer engineering at the Institute of Electronic Design Automation at the Technical University of Munich.

His research interests include logic synthesis for FPGA's and sequential optimization techniques.

**Bernd Wurth** received the Ph.D. and Dipl.-Ing. degrees from the Technical University of Munich, Munich, Germany. His Ph.D. dissertation focused on new logic decomposition techniques for technology mapping to LUT-based FPGA's.

Until recently, he was with Synopsys, Inc., Mountain View, CA, where he developed power analysis and optimization tools. He is now with Siemens Semiconductor Group, Munich, where he is involved in a design reuse methodology project. Previously, he was an Ernst von Siemens Scholar at the Institute of Electronic Design Automation in Munich, where he worked on logic synthesis for FPGA's, power analysis and optimization, test methods, and partitioning.

**Klaus Eckl** received the Dipl.-Ing. degree in electrical and computer engineering at the Technical University of Munich, Munich, Germany, in 1995. He is currently working towards the Ph.D. degree at the Institute of Electronic Design Automation at the Technical University of Munich, Munich, Germany.

His research interests include combinational and sequential logic synthesis with a specific focus on FPGA's.