Tutorial and Survey Paper

Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays

JASON CONG
University of California, Los Angeles
and
YUZHENG DING
AT&T Bell Laboratories

The increasing popularity of the field programmable gate-array (FPGA) technology has generated a great deal of interest in the algorithmic study and tool development for FPGA-specific design automation problems. The most widely used FPGAs are LUT based FPGAs, in which the basic logic element is a K-input one-output lookup-table (LUT) that can implement any Boolean function of up to K variables. This unique feature of the LUT has brought new challenges to logic synthesis and optimization, resulting in many new techniques reported in recent years. This article summarizes the research results on combinational logic synthesis for LUT based FPGAs under a coherent framework. These results were dispersed in various conference proceedings and journals and under various formulations and terminologies. We first present general problem formulations, various optimization objectives and measurements, then focus on a set of commonly used basic concepts and techniques, and finally summarize existing synthesis algorithms and systems. We classify and summarize the basic techniques into two categories, namely, *logic optimization* and *technology mapping*, and describe the existing algorithms and systems in terms of how they use the classified basic techniques. A comprehensive list of references is compiled in the attached bibliography.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles—*combinational logic*; B.6.3 [**Logic Design**]: Design Aids—*automatic synthesis, optimization*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*gate arrays*; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design

General Terms: Algorithms, Design, Experimentation, Measurement, Performance, Theory

Additional Key Words and Phrases: Area minimization, computer-aided design of VLSI, decomposition, delay minimization, delay modeling, FPGA, logic optimization, power minimi-

This work is partially supported by National Science Foundation Young Investigator Award MIP-9357582 and grants from AT&T Microelectronics and Xilinx under the University of California MICRO program.
Authors' addresses: J. Cong, Department of Computer Science, University of California, Los Angeles, CA 90095; email: ⟨cong@cs.ucla.edu⟩; Y. Ding, AT&T Bell Laboratories, Murray Hill, NJ 07974; email: ⟨eugene@research.att.com⟩.

zation, programmable logic, routing, simplification, synthesis, system design, technology mapping.

1. INTRODUCTION

The *field programmable gate-array* (FPGA) is a new technology evolved in the last decade as an alternative to application-specific integrated circuit (ASIC) designs. An FPGA is an off-the-shelf VLSI chip consisting of programmable logic elements, programmable I/O elements, and programmable routing elements. Designers can realize their designs by *programming* the FPGA chips in the field, thus avoiding the often lengthy and expensive fabrication process. An important FPGA technology is the *static random-access memory* (SRAM) based FPGA, in which programmability is realized by using SRAM cells to implement programmable logic elements and to control programmable routing elements. Because contents of SRAM cells can be rewritten by the user, the SRAM based FPGAs have the advantage of *field-reprogrammability*, which leads to many important applications. The most common approach of implementing the basic logic element in an SRAM based FPGA is via a 2^K-bit SRAM cell, which represents a K-input one-output *lookup-table* (LUT), capable of realizing any Boolean function of up to K variables by loading the SRAM cell with the truth table of that function. Such LUT *based* FPGAs are available from several major commercial FPGA vendors, such as Altera [1994], AT&T [1995], and Xilinx [1994]; and are so far the most widely used type of FPGAs.

The FPGA design process is similar to that for conventional technologies such as standard cell or gate array. It consists of the system-level design (high-level synthesis), the logic-level design (logic synthesis), and the physical-level design (layout synthesis). The logic-level design for FPGAs consists of *sequential logic synthesis*, which optimizes the assignment and/or arrangement of storage elements, and *combinational logic synthesis*, which optimizes the logic gate networks. This article focuses on the combinational logic synthesis problem for LUT based FPGAs. Sequential logic synthesis techniques for FPGAs, such as retiming (Malik et al. [1991], Pan and Liu [1996], Touati et al. [1992], and Weinmann and Rosenstiel [1993]), state-splitting (e.g., Allen [1992]), and flip-flop assignment (e.g., Murgai et al. [1993]) are not covered in this article.

Conventional library based synthesis techniques, such as those in Detjens et al. [1987], Devadas et al. [1994], and Keutzer [1987], are difficult to apply to LUT based FPGA synthesis directly, as a K-input LUT can implement 2^{2^K} different functions,[1] which vary significantly in terms of network size and depth when represented as networks of basic logic gates.

[1] If we consider the equivalent classes under input permutation, the number is reduced, but is still very large for $K > 4$.

This problem has motivated much research on combinational logic synthesis techniques specifically for LUT based FPGAs, and many results have been published in the past few years. These results, however, were dispersed in a large number of papers in various conference proceedings and technical journals, presented in different terminologies and often in architecture-specific contexts, making it difficult to follow the progress in this field. Although an early survey paper [Sangiovanni-Vincentelli et al. 1993] and several books (e.g., Brown et al. [1994] Chan and Mourad [1994], Murgai et al. [1995], and Trimberger [1994]) have provided good introductions to FPGA technologies and FPGA synthesis techniques, they do not cover many results obtained more recently. Given the rapid growth of this field, there is a strong need for a comprehensive survey of FPGA synthesis results to include up-to-date information. This article is written to fulfill such a need.

Instead of tracing the evolution of the theory and practice in this field, this article focuses on presenting the commonly used *basic techniques* in a coherent framework. The reason is that due to the complexity of the problems, almost every study in this field has used multiple techniques and aimed for multiple objectives. Most techniques are shared by multiple FPGA synthesis algorithms and systems. Identification and understanding of these basic techniques are helpful not only for the understanding of the complete algorithms and systems that were built upon them, but also for the development of new algorithms and systems. Therefore we first present these basic techniques in an algorithm/system-independent fashion, and then review the complete algorithms and systems in terms of how these basic techniques were used.

There are several possible criteria to classify these basic techniques (or *operations*). We first recognize the two major steps of combinational logic synthesis, namely, *logic optimization*, which transforms the gate-level network into another equivalent gate-level network which is more suitable for the subsequent step, and *technology mapping*, which transforms the gate-level network into a network of logic elements (or *cells*) in the target technology by covering the network with the cells. This distinction is used to classify the basic techniques in our article. We choose to follow this classification because many synthesis algorithms and systems indeed go through these two steps. Also, the techniques used in these two steps generally have different characteristics. Logic optimization operations usually rely on the knowledge of the *functionality* of the gates and the network, and use *Boolean optimization* techniques, whereas technology mapping operations usually depend heavily on the *structural information* of the gates and the network, and use *combinatorial optimization* techniques. Such a dividing line is not always easy to draw—some logic optimization operations may use combinatorial methods along their course, and some technology mapping operations also make use of the gate and network functionality. In some FPGA synthesis systems, a separate mapping or covering step does not even exist, as a gate-level network with bounded gate input can be viewed as an LUT network.

Another classification of FPGA synthesis algorithms, often used in literature, is based on their optimization objectives, such as area minimization, delay minimization, routability optimization, and so on. We choose not to use this classification because many optimization techniques can be used for multiple objectives with proper choice of cost functions. We point out the applicable objective(s) when presenting each technique in later discussions.

The rest of this article is organized as follows. Section 2 presents the basic concepts, terminologies, and problem formulations. Section 3 presents the basic techniques for logic optimization, with emphasis on various node decomposition techniques. Section 4 presents technology mapping methods. Then, in Section 5, we briefly summarize various combinational logic synthesis algorithms for LUT based FPGAs published in recent literature, in terms of their choice and combination of the techniques presented in Sections 3 and 4. A bibliography of related publications is compiled at the end of the article. To keep the article concise, most proofs and algorithmic details are omitted. Although experimental results as well as comparative studies were reported for most algorithms in their original publications, they are not always on the same ground due to different choices of the benchmark examples, the initial optimization procedures, and various algorithm-specific and/or architecture-specific parameters used. This prevents us from providing a fair across-the-board quantitative comparison of all the techniques. We also avoid citing the reported experimental results directly, except in a few cases where the consistency of the experimental environments for comparison was guaranteed.

2. PRELIMINARIES AND PROBLEM FORMULATIONS

2.1 Design Representation

Inasmuch as our interest is in combinational logic synthesis, we focus on the combinational portion of a design, which is assumed to be a two- or multi-level network of logic gates. It carries two types of information, namely, the *structural information* and the *functional information*.

2.1.1 *Representation of Structural Information.* A combinational circuit can be viewed as a directed acyclic graph (DAG) $N = (V(N), E(N))$, where each node $v \in V(N)$ represents a logic gate or a primary input/output, and a directed edge $(v, w) \in E(N)$ indicates that the output of gate v is an input of gate w, in which case v is called a *fanin* of w and w is a *fanout* of v. A primary input (PI) node has no fanin and a primary output (PO) node has no fanout, and a node with both fanins and fanouts is an *internal* node. The PI and PO nodes are created to carry the outputs and inputs of noncombinational logic elements, such as flip-flops and I/O pads, respectively; the internal nodes represent the logic gates. The *level* of a node v is the maximum number of edges on any path from a PI to v, and the *depth* of the network is the largest level of its primary outputs. If there is a path from node v to node w, v is a *predecessor* of w and w is a *successor* of v. Because the network is a DAG, a partial order called *topological order*

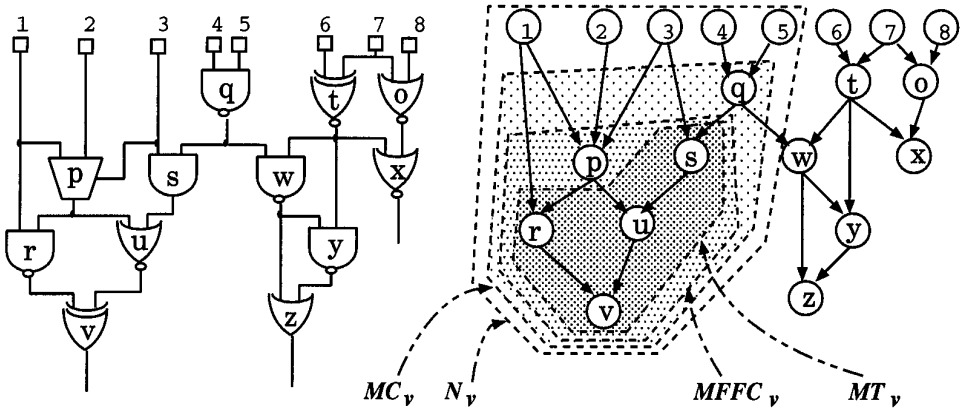

Fig. 1. A multilevel logic gate network, its DAG network, and the fanin network, the maximum tree, and the maximum fanout-free cone of v.

exists among the nodes, such that each node appears after all its predecessors and before all its successors in the ordering.

Given a network N, the set of fanins of gate v is denoted $input(v)$, and the set of *distinct* nodes that supply inputs to the gates in subnetwork H is denoted $input(H)$. A node v is *K-feasible* if $|input(v)| \leq K$, and a subnetwork H is *K-feasible* if $|input(H)| \leq K$. If every node in a network is K-feasible, the network is *K-bounded*. Similarly, the set of fanouts of v is denoted $output(v)$, and the set of fanouts of a subnetwork H is denoted $output(H)$. A node v is *fanout-free* if $|\, output(v)\,| \leq 1$. If every nonPI node in a network is fanout-free, the network is *internal fanout-free*, and is called a *leaf-DAG*. If every node (including PI) is fanout-free, the network is called a *tree* if it has a single PO node, or a *forest* if it has multiple PO nodes.

There are several useful substructures in a network. Given a node v in the network N, a *cone of* v, denoted C_v, is a subnetwork of N consisting of v and *some* of its nonPI predecessors such that for any node $w \in C_v$, there is a path from w to v that lies entirely in C_v. Node v is called the *root* of C_v. The *maximum cone* of v consisting of *all* the nonPI predecessors of v is denoted MC_v. The *fanin network* of v, denoted N_v, is an extension of the MC_v by including all of the PI predecessors of v as well.

A *fanout-free cone* (FFC) is a cone in which the fanouts of every node other than the root are in the cone (i.e., they *converge* to the root). For each node v there is a unique *maximum fanout-free cone* (MFFC) [Cong and Ding 1993a] of v, denoted $MFFC_v$, which contains every FFC rooted at v. An equivalent characterization is $output(w) \subseteq MFFC_v$ implies $w \in MFFC_v$ or w is a PI. An FFC is not necessarily a tree, but any subnetwork that is a tree forms an FFC. For each node v, there is also a *maximum tree* (MT) of nonPI nodes rooted at the node, and is denoted MT_v. Clearly, $MT_v \subseteq MFFC_v \subset N_v$. The K-feasibility for various cone or tree structures is defined the same way as that for the general subnetworks. Figure 1 shows a multilevel network of logic gates, its DAG network representation, and

various structural elements. A topological ordering is (1, 2, . . . , 8, o, p, q, . . . , z).

A cone C_v defines a bipartitioning $(X, \bar{X})$ of the nodes in N_v with $\bar{X} = C_v$, such that $v \in \bar{X}$ and all PIs are in X. This bipartitioning is called a *cut* of N_v, and the *size* of the cut is defined as $| input(\bar{X}) |$, where $input(\bar{X})$ is the *node cut-set*. Similarly, we define a cut of a cone C_v to be a bipartitioning $(X, \bar{X})$ of the nodes in $C_v \cup input(C_v)$ such that $input(C_v) \subseteq X$, $v \in \bar{X}$, and $\bar{X} \subseteq C_v$ is a subcone. A cut is *K-feasible* if its size is no more than K, that is, $\bar{X}$ forms a K-feasible cone. Because the node cut-set $input(\bar{X})$ uniquely determines the cut, it is often referred to as the cut itself.

A general network can be partitioned into a collection of subnetworks. The *cone partitioning* [Saucier et al. 1993a] divides a network into the fanin networks of its primary outputs. Clearly, cone partitioning is not a *disjoint* partitioning. Two important disjoint partitionings are the *tree partitioning* [Keutzer 1987] and the *MFFC partitioning* [Cong and Ding 1993a].[2] Given two nodes v and w, the two MTs MT_v and MT_w are either disjoint or one contains another. Similarly, the two MFFCs $MFFC_v$ and $MFFC_w$ are either disjoint or one contains another. Therefore the set of MTs (or MFFCs) that are not contained in other MTs (MFFCs) defines a disjoint partitioning, which can be obtained as follows: first, the MT (MFFC) of each PO is a partition; then recursively the MT (MFFC) of each input node to an existing partition is also a partition. Note that the input nodes to a partition are not part of the MT (MFFC), and may have multiple fanouts to the nodes inside and/or outside the partition. In a tree partitioning, a tree and its input nodes often need to be considered together during logic synthesis. In this case, they form a leaf-DAG.

2.1.2 *Representation of Functional Information.* Each gate in a multilevel gate network and, correspondingly, each internal node v in the DAG representation, is associated with a Boolean function f_v in terms of the signals from $input(v)$. It can be one of the *simple-gate* functions—inverter (INV), and (AND), inclusive-or (OR), exclusive-or (XOR), and their complements (NAND, NOR, XNOR); or any *complex-gate* function. We first review various representations of a Boolean function.

A single-output Boolean function f can be defined by partitioning its input vectors into the ON set ON_f, the OFF set OFF_f, and the DC set DC_f, corresponding to the vectors that evaluate the function to 1, 0, and the undefined value (called *don't-care*). If $DC_f = \varnothing$, f is *completely specified*; otherwise it is *incompletely specified*. For a completely specified function f, only ON_f needs to be explicitly specified. An incompletely specified function f can be represented by two completely specified functions g and h, with $ON_g = ON_f$ and $ON_h = DC_f$. A straightforward way of representing f is to use the *truth table* which lists all input vectors and the corresponding function values. Figure 2(a) shows an example of a 2-D truth table.

[2] Originally they were called *tree decomposition* and *MFFC decomposition*, but we changed the names to avoid possible confusion with the decomposition of gates we discuss later.

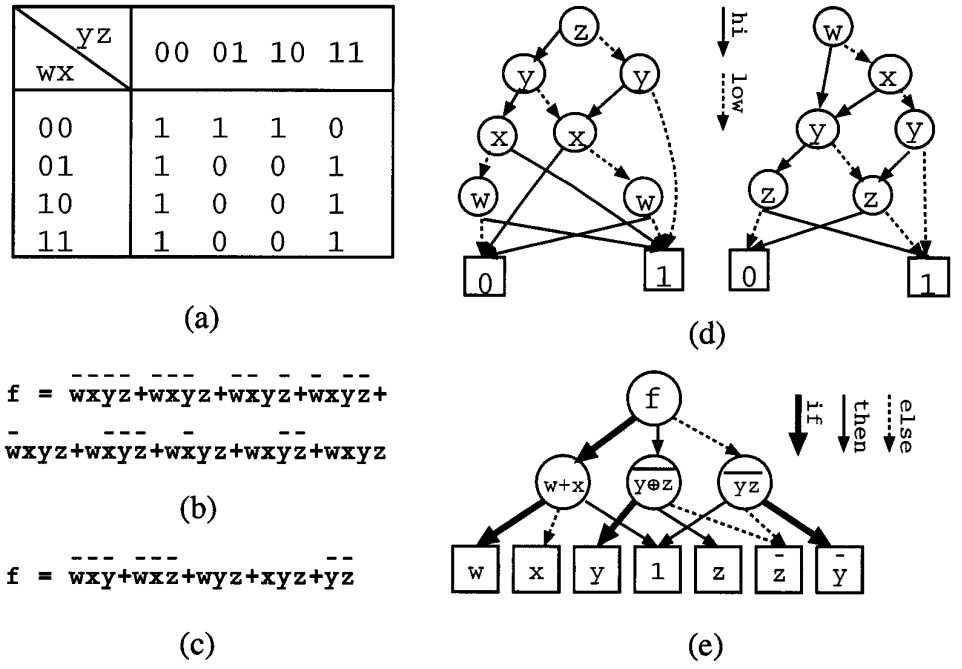

Fig. 2. Various representations of a completely specified function: (a) 2-D truth table; (b) minterm SOP; (c) cube cover; (d) two OBDDs with different variable orders; (e) ITE.

A *literal* is a variable or its complement. A *cube* is a product term (AND) of some literals. A *minterm* is a cube in which the literal of every variable appears exactly once. Each minterm corresponds to a bit vector in the input vector space (e.g., $a\bar{b}c$ corresponds to 101), and a cube contains one or more vectors (e.g., $a\bar{b}$ contains $a\bar{b}c$ and $a\bar{b}\bar{c}$). A set of vectors can then be represented by a collection of minterms in the form of *sum-of-products* (SOP). A *cube cover* is a collection of cubes whose union covers ON_f but does not intersect OFF_f. Cube cover is one of the most popular ways to represent a function. More details on its manipulations can be found in Brayton et al. [1984]. The DC set of an incompletely specified function can be used for cube cover minimization, as a cube cover can be extended to cover not only all the vectors in the ON set, but also some or even all of the vectors in the DC set. Figures 2(b) and 2(c) show the minterm SOP and a cube cover (also in SOP form), respectively, of the function defined in Figure 2(a).

The *decision diagrams* are another type of popular logic representation, led by the *binary decision diagram* (BDD) first introduced in Akers [1978]. A BDD is a rooted DAG that consists of a *root* with no incoming edge, two *terminals* with no outgoing edge, and possibly other nonterminal vertices. Each nonterminal vertex v is associated with a variable $var(v)$ of the function, and has exactly two outgoing edges marked 1 and 0, whose destination vertices are called $hi(v)$ and $low(v)$, respectively; the two terminals are marked 1 and 0. Such a DAG, as well as each of its rooted subgraphs, represents a completely specified function, defined recursively

as follows. The terminal vertices marked 1 and 0 represent constant functions 1 and 0, respectively, and a nonterminal vertex v represents the function $f_v = var(v) \cdot f_{hi(v)} + \overline{\text{var}(v)} \cdot f_{low(v)}$. (This recursion is called *Shannon expansion*.) The most famous type of BDD is the *reduced ordered BDD* (ROBDD, or OBDD in short) by Bryant [1986]. In an OBDD, the variable ordering along every path from the root to a terminal must be the same, and each vertex must be unique and irredundant. That is, for any nonterminal vertex v, $hi(v) \neq low(v)$, and for any two distinct vertices u and v, if $var(u) = var(v)$, then either $hi(u) \neq hi(v)$ or $low(u) \neq low(v)$. Given a variable ordering, OBDD is a *canonical* representation; moreover, functions under OBDD representation can be manipulated easily under Boolean operations and many other transformations. More discussions on OBDD and its manipulations can be found in Bryant [1986, 1992]. An efficient implementation of an OBDD package was described in Brace et al. [1990].

OBDD representation can be more compact than cube based representation due to the sharing of its irredundant subOBDDs and its multilevel representation. Further size reduction can be achieved by sharing subOBDDs among multiple functions (called *multiroot* OBDD), and by allowing *complemented edge* so that subfunctions f and $\overline{f}$ can be represented by the same subOBDD [Madre and Billon 1988]. (Complemented edges are restricted only to the *low* edges to ensure a canonical representation.) In the worst case, however, an OBDD can still have an exponential size in terms of the number of variables. The size of an OBDD is strongly related to the variable ordering: Figure 2(d) shows two OBDDs of the same function using different variable orders. Many studies have been focused on obtaining a good ordering that minimizes the OBDD size (see, e.g., Panda et al. [1994], Rudell [1993], and Soe and Karplus [1993]).

BDDs can be easily converted to multilevel logic gate networks. Each nonterminal vertex v can be converted into a 2-to-1 multiplexor (a MUX2 gate), with $var(v)$ as the selector input, and $hi(v)$ and $low(v)$ as two data inputs. The terminals become constant inputs, and are propagated throughout the network for simplification. BDDs can also be extended to handle incompletely specified functions by adding a terminal for don't-care (e.g., see Matsunaga and Fujita [1989]).

There are also many other decision diagrams (see Bryant [1995] for a partial list). One of them is the *If-Then-Else DAG* (*ITE*) [Karplus 1989], in which each nonterminal vertex v has three outgoing edges $i(v)$, $t(v)$, and $e(v)$, and a terminal can be associated with a constant or a literal. Each nonterminal vertex is associated with the function $f_v = f_{i(v)} \cdot f_{t(v)} + \overline{f_{i(v)}} \cdot f_{e(v)}$, which reads *if* $i(v)$ *then* $t(v)$ *else* $e(v)$. Figure 2(e) shows an example. ITE is more concise than BDD [Lam and Brayton 1992], although it is also more difficult to manipulate and come up with a canonical representation.

Both cube cover and BDD can be used to represent a node function f_v, which is associated with node v using $input(v)$ as the variable set. We can define other functions at node v as well. Given a cone C_v rooted at v, we use

f_{C_v} to denote the function using $input(C_v)$ as the variable set; that is, f_{C_v} is the function associated with a cone C_v. In particular, f_{MC_v} is the function associated with v with respect to the primary inputs. If $v_1, \ldots, v_m$ are the primary outputs, then the functionality of the network is defined by $\{f_{MC_{v1}}, \ldots, f_{MC_{vm}}\}$, the functions of the primary outputs with respect to the primary inputs. Function f_{C_v} can be derived by composing the node functions in C_v.

2.2 Combinational Logic Synthesis for LUT Based FPGAs

Given a multilevel network of logic gates, combinational logic synthesis transforms it into a network of LUTs, each of no more than K inputs (denoted K-LUT) where K is determined by the target FPGA technology.[3] (Therefore it is referred as *LUT logic synthesis* in the rest of the article.) This transformation usually includes two major steps, the *logic optimization* and the *technology mapping*. Logic optimization works on a network of logic gates, and transforms it into another network of logic gates suitable for mapping into a network of LUTs. Technology mapping then covers the network with K-LUTs. (For simplicity, we reserve the symbol K as the maximum input size of an LUT, and use LUT to indicate K-LUT in the rest of the article.)

These two steps are not clearly distinguished in a number of LUT logic synthesis algorithms. Because each gate of no more than K inputs can be implemented by an LUT, a separate technology mapping step may be skipped if the result of logic optimization is a K-bounded network. Iterations over these two steps are also quite common where a mapped network is further optimized as a gate network and then remapped. However, techniques for these two steps are generally different in nature. Logic optimization relies on the network functionality, whereas technology mapping depends on the network structure. Therefore we choose to present the techniques used in these two steps separately.

2.2.1 *Logic Optimization.* The goal of logic optimization is to transform the given network into an equivalent optimized network which is more suitable for mapping into LUTs. The minimum requirement for this is that the resulting network must have a valid K-LUT mapping solution, in which case the network is said to be *K-mappable*. It is not hard to show that the *sufficient and necessary* condition for a multilevel network to be K-mappable is that every node has a K-feasible cone.[4] Because for a given network there are many equivalent K-mappable networks, the goal of logic optimization is to produce an equivalent network that will have a *good* mapping

[3] Although many commercial FPGA architectures support LUTs of different sizes on one chip (e.g., the Xilinx XC4000 [Xilinx 1994] supplies 3-LUTs and 4-LUTs, which can also be combined to form 5-LUTs; the AT&T ORCA2C [AT&T 1995] supplies 4-LUTs that can be combined to form 5-LUTs and 6-LUTs), most studies assume a homogeneous LUT architecture with uniform LUT size for simplicity.

[4] Note that a K-bounded network is K-mappable, but a K-mappable network does not have to be K-bounded.

solution according to one or several mapping objectives. Unfortunately, due to the flexibility in LUT mapping, there has been no good quantitative measurement of logic optimization operations for LUT based FPGAs without going through the mapping process. As a result, logic optimization operations often aim for intuitively good solutions, such as those with a small number of gates, small gate input size, small depth, and sparse interconnections.

Most techniques used for logic optimization in LUT logic synthesis have their roots in multilevel combinational logic synthesis for library based technologies, such as standard cell or gate array designs. Summaries of those techniques in the context of conventional logic synthesis can be found in Brayton et al. [1990], Devadas et al. [1994], and DeMicheli [1994].

We classify the logic optimization techniques for LUT into two categories, namely, *node decomposition* and *network simplification*. Node decomposition extracts subfunctions from the functions of one or more nodes, and creates new nodes to implement these subfunctions, resulting in a representation with more but simpler nodes (e.g., with fewer fanins). Network simplification reexpresses the functions of one or more nodes and modifies the interconnection accordingly, resulting in a simpler network with fewer nodes, sparser interconnections, and/or smaller network depth. Details of logic optimization techniques are presented in Section 3.

2.2.2 *Technology Mapping.* The task of technology mapping in LUT logic synthesis is to cover the optimized gate-level network with LUTs. (For simplicity, we also call it *LUT technology mapping* or *LUT mapping* for short.) We say an LUT LUT_v *implements* a node v if LUT_v covers a cone whose root is v. A valid covering must satisfy the following conditions: (1) every PO node is an output of an LUT; and (2) if v is implemented by LUT_v, then every nonPI node in *input* (LUT_v) must also be implemented by some LUT. An *irredundant* covering implements only the nodes defined by these conditions. Nodes that are not implemented are covered entirely in the LUTs implementing their successors. Throughout this article, we always assume that the input network to the technology mapping step is K-mappable so that a mapping solution is always possible.[5]

Note that LUTs may overlap, which implies that the overlapped portion of logic will be duplicated into each of these overlapping LUTs. When no logic duplication is allowed, the mapping is called a *duplication-free mapping*. Figure 3 illustrates various types of mappings of a gate-level network.

Although LUT covering can be viewed as a procedure of collapsing-based network simplification, the procedure itself does not care about the functionality of each covered cone. Therefore technology mapping operations are combinatorial in nature and work on the structural representation of

[5] Although an integrated logic optimization and technology mapping process can take an unbounded network as input, its logic optimization operations will make the portion of network to be mapped K-mappable prior to its covering.

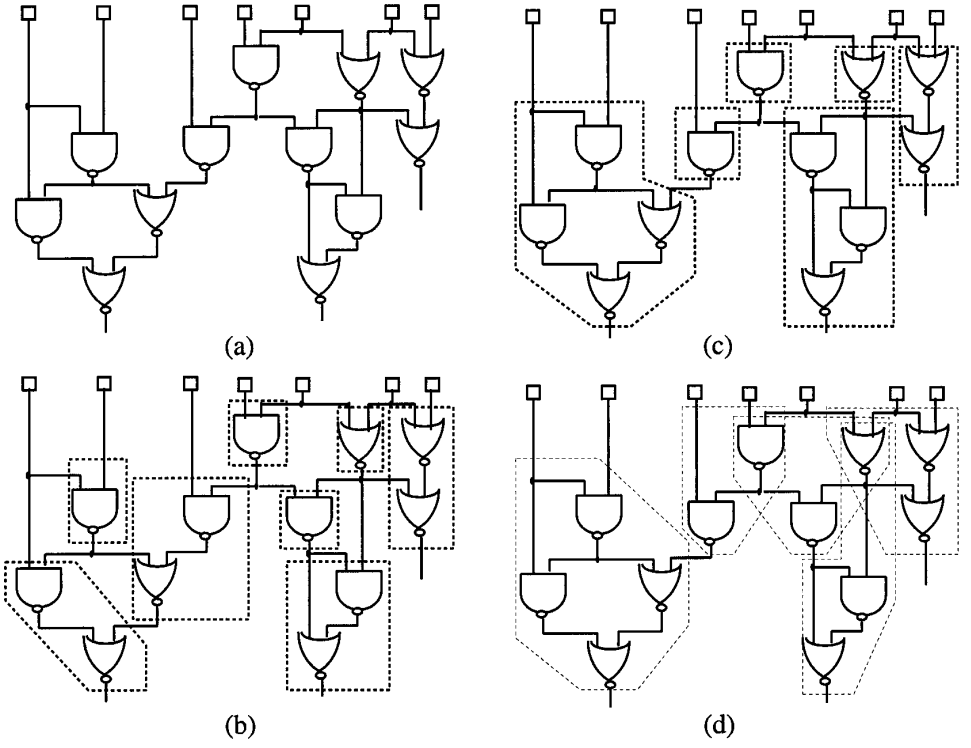

Fig. 3. Various LUT mappings of a network ($K = 3$): (a) original network; (b) mapping after tree partitioning; (c) duplication-free mapping; (d) general mapping with overlapping LUTs.

the network. Details of the technology mapping methods are presented in Section 4.

2.3 Optimization Objectives and Optimality

There are several optimization objectives for LUT logic synthesis. The *area minimization* objective is to use the minimum chip area to implement the network; the *delay minimization* objective is to minimize the delay from primary inputs to primary outputs in the FPGA implementation; the *routability optimization* objective is to make the LUT network more routable in the subsequent placement and routing steps; and the *power minimization* objective is to minimize the power dissipation of the implementation. Accurate measurement for these objectives requires information that is only available *after* layout synthesis. Therefore various estimations are used during logic synthesis, either by a quick layout synthesis, or by simplified cost models. We introduce some commonly used cost models in the following.

Given an LUT network, its area is usually estimated by the number of LUTs (or actual logic elements if aiming for a specific FPGA architecture)

used for its implementation. Although a close approximation, it may not be accurate, as other constraints, such as flip-flop distribution and routing congestion, may prevent 100% utilization of logic elements.

The delay of an LUT network is measured by the length of the longest path from a primary input to a primary output, where the *length* is the accumulation of the delays associated with the nodes and edges along the path. For a given technology, the LUT delay is roughly a constant; therefore the various delay models focus on the net delays. The most commonly used model (For example, see Cong et al. [1992a], Cong and Ding [1992], Francis et al. [1991a], and Shin and Kim [1995], among others) is the *unit delay model*, which assumes that each net has a constant delay. Under the unit delay model, delay minimization is equivalent to *depth* minimization. Improvements over the unit delay models include the *net delay model* [Cong et al. 1993], which allows each net to be assigned a different delay value, and the *edge delay model* [Yang and Wong 1994], which allows each branch (*edge*) of a net to be assigned a different delay value. The delay values should reflect the impact of placement and routing on each net (or branch). In general, a *static delay model* assigns the delay values to each net (or branch) prior to mapping, and assumes that the delay of that net (branch) will remain unchanged in the final mapping solution if it is still visible (i.e., is an output of an LUT), and become zero if it is not visible (i.e., covered inside LUTs). This type of models faces the problem that the delay values are difficult to determine prior to mapping.

Alternatively, one can estimate the delay value for a net (branch) based on its structure in the *mapped network*. For example, the *nominal delay model* [Cong and Ding 1994a; Schlag et al. 1991] assumes that the net delay is proportional to net size in the mapped solution. Inasmuch as nominal delay is related to the structure of the mapped network, it is difficult to predict *prior to* mapping. In general, a *dynamic delay model* directly associates a function (of the net structure) with each net as its estimated delay, which will change as the net structure changes. Although more accurate, dynamic delay models are more difficult to use in optimization.

Routability is usually modeled by interconnection complexity, in particular, the size and the terminal distribution of the net. Simplified measurements include the *pin-to-net ratio* and *pin-to-cell ratio* [Schlag et al. 1992]. Power minimization is of recent interest in logic synthesis. The power consumption can be estimated based on the output load capacitance and transition frequency of the LUTs and primary inputs. The load capacitance changes dynamically during the mapping process depending on the output net structure [Farrahi and Sarrafzadeh 1994a].

In general, these optimization objectives may not be mutually compatible. For example, minimizing the number of LUTs may reduce the parallelism of the design, resulting in larger delay [Cong and Ding 1994b]. Therefore it is often necessary to find a proper tradeoff among different objectives. Because FPGA chips are prefabricated, area and routability optimizations usually only need to pursue a *feasible* solution that fits into

the chip. Delay minimization, on the other hand, should aim at the *best* solution, under the area and routability constraints. Most studies on FPGA logic synthesis have focused on area and delay minimization, and so does our discussion.

3. LOGIC OPTIMIZATION

In this section we introduce the techniques, used in the logic optimization step, that transform a multilevel network of generic gates into another K-mappable gate-level network which is more suitable for LUT implementation. Node decomposition techniques presented in Sections 3.1–3.4 are our focus, and network simplification techniques are discussed in Section 3.5.

Node decomposition reexpresses a node function by a logically equivalent composition of two or more functions, and correspondingly introduces two or more new nodes to replace the original node. It is an important type of logic optimization operation, especially for LUT logic synthesis, as the original network may not be mappable. If a node v does not have a K-feasible cone, some of the nodes in N_v have to be decomposed to reduce their input sizes. In addition to making a network mappable, reducing node input size also gives technology mapping more freedom to cover (or *pack*) the nodes into LUTs. Moreover, node decomposition is more flexible in LUT logic synthesis, because as long as the input size of a decomposed node is no more than K, it can be implemented by a K-LUT regardless of its complexity. As a result, some previously developed, but not widely used, techniques (such as functional decomposition) find important applications in LUT logic synthesis.

There are various node decomposition techniques for various kinds of node functions and optimization objectives. It can be performed towards one or more nodes, for simple- or complex-gate functions, and based on different form of representations. We classify them into three types: *structural decompositions* that apply to simple gates or certain simple-gate networks, *symbolic decompositions* that apply to complex gates based on symbolic operations on a given form of functional representation, and *Boolean decompositions* that apply to complex gates based on their general functionality. We also introduce several simple bounds on the optimality of decomposition.

3.1 Structural Decomposition Methods for Simple Gates

Simple-gate functions (AND, OR, XOR and their complements) have *commutative* and *associative* properties, which allow arbitrary grouping of the inputs in decomposition. As a result, knowledge of the specific gate function is not required, and the gate decomposition becomes a *combinatorial* operation. In this subsection, we consider the decomposition of simple gate v with $input(v) = \{w_1, \ldots, w_m\}$ $(m > K)$ into a tree of nodes of input size K or smaller.

3.1.1 *Balanced Tree Decomposition.* The *balanced tree decomposition* divides $input(v)$ into K groups of equal (or nearly equal) size, introduces K new gates, each carrying inputs from one group of $input(v)$, and replaces $input(v)$ with the set of these K new gates. Each new gate is recursively decomposed if necessary. This method has been widely used (e.g., as part of the tech_decomp command of the MIS-II logic synthesis system [Brayton et al. 1987, and in logic optimization for LUT logic synthesis [Schlag et al. 1992]).

This approach results in a decomposition tree of minimum depth, but the size of the decomposition tree (i.e., the number of gates used in the decomposition) may not be minimum (when $K > 2$) due to the balance constraint. Without the strict balance constraint, minimum depth and minimum gate count can be achieved simultaneously using the following *minimum tree decomposition* method. A list L of nodes currently in $input(v)$ is maintained in the FIFO order. In each iteration, the first K nodes in L are removed, and a new node is created with these K nodes as inputs and added back to L. This procedure ends when $| L | \leq K$. This procedure will result in $D = \lceil (m - 1)/(K - 1) \rceil$ gates (including v), which is the *minimum*. The heights from the root to different leaf nodes differ at most one. The complexity of this operation is $O(m)$.

3.1.2 *Huffman Tree Decomposition.* If the inputs of v have different levels (or arrival delays), balanced tree decomposition does not yield minimum depth (or delay) at the root: According to Chen et al. [1992], when applied to a simple-gate network of depth d, balanced tree decomposition may increase the depth to $d \log d$.

Because the inputs removed earlier from L will have larger distances from the root, one can modify the minimum tree decomposition algorithm to maintain L as a sorted list in the nondecreasing order of the levels of the nodes, thus the nodes of the smallest levels are always removed first. This method, regarded as *Huffman tree decomposition* due to its similarity to the *Huffman encoding* idea [Huffman 1952] for data compression, has been used in Cong et al. [1992b] and Wang [1989]. This method gives a delay optimal decomposition of the entire simple-gate network if applied in topological order from PIs to POs. The optimality can be generalized to the case of nonunit delay models as well. Moreover, it was shown [Chen et al. 1992] that after the Huffman tree decomposition, the network depth will be bounded by $\log (2f) \cdot d + \log I$, where f is the maximum fanout of any node (usually a small constant) and I is the number of primary inputs. The time complexity of this operation is $O(m \log m)$ due to the need of maintaining a sorted list L.

3.1.3 *Bin-Packing Decomposition.* The Huffman tree decomposition produces trees of the minimum size and depth. However, such trees may not result in minimum size or depth LUT mapping, as an LUT may cover a number of nodes in one or several decomposition trees. This problem is overcome by the *bin-packing based decompositions* [Francis et al. 1991a,b] when the network is a tree of simple gates.

Decomposition for Area Minimum LUT Covering. The bin-packing based decomposition is carried out in topological order from PIs to POs such that when node v is to be decomposed, every node $w \in input(v)$ is either a primary input or a node that is already decomposed, and hence is K-feasible. The objectives at each node v are (1) to decompose v into a tree T_v of nodes of K or fewer inputs rooted at v' (which replaces v), such that the number of K-LUTs needed to cover T_v, *as well as* $input(v)$, is minimum, and (2) $|input(v')|$ is minimum under condition (1). It was shown in Francis et al. [1991b] that if these two conditions are satisfied, the decomposition will have a minimum number LUT covering for a tree.

In order to satisfy the two conditions, the problem is formulated as the *bin-packing* problem,[6] where the LUTs are bins of size K, and each node $w \in input(v)$ is an object of size $input(w)$. The algorithm is as follows. First, each node $w \in input(v)$ is packed into bins (LUTs) in turn by *restricted* bin-packing, which will choose among all the minimum bin solutions the one that has the least saturated bin [in order to satisfy condition (2)]. Then, repeatedly the bin B with maximal $input(B)$ is closed, a new node u is created with $input(u) = B$, and packed as a new object of size 1 into the unsaturated bin that is most full (or if none remains, a new bin). This procedure ends when there is only one bin left, which creates the root node v'. It can be shown that the result of this algorithm satisfies the two conditions listed, thus giving each node a decomposition such that they can be covered totally by the minimum number of LUTs. Although the bin-packing problem is NP-hard [Garey and Johnson 1979], in practice K is not too large, and an exact algorithm may be affordable. Heuristic bin-packing algorithms [Johnson et al. 1974] can also be used. In particular, the *first-fit decreasing* (FFD) algorithm, which sorts the objects in decreasing order of sizes and packs each object to the first bin that has enough room, was shown to be optimal for the restricted bin-packing problem when $K \leq 5$ [Francis et al. 1991b]. The *best-fit decreasing* (BFD) algorithm, which is similar to FFD but packs each object into the bin that has the least available room after accommodating the object, also has the same property [Murgai et al. 1991b]. In either case, the heuristic bin-packing can be implemented with time complexity of $O(m \log m)$. Figure 4 illustrates the bin-packing decomposition of a node using the FFD heuristic, where the objects $[p, s]$ and $[q]$ are packed into one bin, $[r]$ and $[t]$ into another, and $[u]$ into the third one; this derives a decomposition of v.

Decomposition for Delay Minimum LUT Covering. A similar approach can obtain a decomposition for delay minimum LUT covering [Francis et al. 1991a]. We present the idea using the unit delay model, which is also applicable to other delay models.

[6] The bin-packing problem is to pack a set of *objects*, each of an integer size $\leq B$, into the minimum number of *bins* of size B such that the total size of the objects in each bin is no more than B. Algorithms for bin-packing can be found in Garey and Johnson [1979] and Johnson et al. [1974].

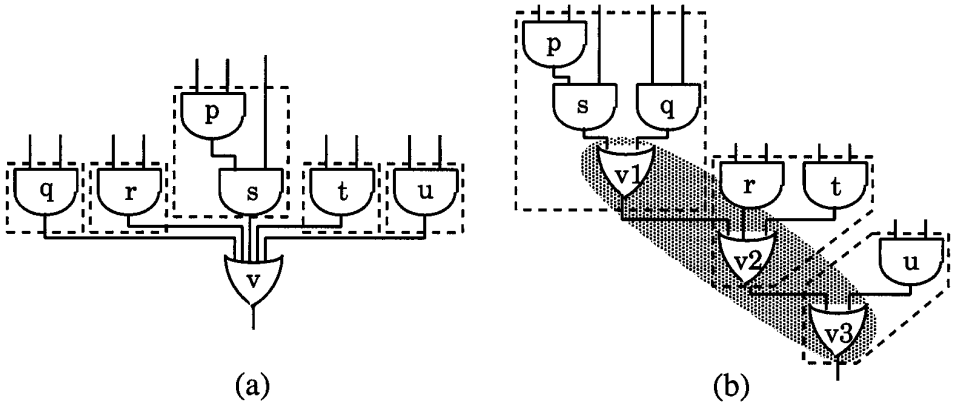

Fig. 4. Bin-packing decompositions at node v of a tree for 5-LUT mapping. Dashed boxes indicate anticipated LUTs: (a) before decomposition; (b) after decomposition.

It was shown [Francis et al. 1991a] that if a decomposition procedure of a tree can yield a covering of the minimum number of LUTs *at each level* in the covering solution, the resulting LUT covering has the minimum depth. To achieve this minimization, bin-packing based decomposition is carried out at each level in the increasing order. Each bin used at the current level becomes an object of size one at the next level, and the procedure ends when there is only one object left at depth $d + 1$, which implies a single bin at depth d, that is, the root of the decomposition tree. Again, the FFD heuristic algorithm can be used. It achieves an optimal solution for $K \leq 6$, a better result than area minimum covering, because the restriction on optimal bin packing is not required [Francis et al. 1991a].

Enhancements. To minimize delay in the area minimum decomposition, in the second phase of bin-packing when several bins can be closed, the one with minimum delay should be closed first. To reduce area in the delay minimum decomposition, some bins at the previous levels can be unpacked and their objects repacked into the unsaturated bins at the current level [Francis et al. 1991a].

The bin-packing based decompositions are optimal only for trees. The formulation is invalid when the network is a leaf-DAG or general network, because the combined size of two or more objects may not be the sum of their sizes when the nodes share fanins. Inasmuch as tree partitioning produces leaf-DAGs, it is important to generalize the bin-packing based decomposition to leaf-DAGs. An exhaustive search approach [Francis et al. 1991b] considers for a node v all possible K-feasible pregroupings of nodes in $input(v)$ that share inputs. Each group is packed together during bin-packing as a single object, using their combined size as the object size. For a given pregrouping, the solution may still be suboptimal. But if all possible groupings are tried, the optimal solution is guaranteed to be found.

Heuristics were also proposed as more practical alternatives. The *maximum share decreasing* (MSD) heuristic [Francis et al. 1991b] is an exten-

sion of the FFD heuristic. At each iteration, MSD chooses one of the largest objects that also has the largest number of shared inputs with any bin and the largest number of shared inputs with any other object, and packs it into an available bin that shares the most inputs with it. The number of shared inputs with an unpacked object can be used to break ties. Similarly, two extensions of the BFD heuristic were proposed in Murgai et al. [1991b], by different choices of the "best" bin: in the *minimum support* heuristic, the best bin is a nonempty one that is most unsaturated after packing the current node, whereas in the *minimum support increment* heuristic, the best bin is a nonempty one whose capacity will be reduced the least after the packing (i.e., the bin that shares the maximum number of inputs with the node to be packed). Another method, proposed in Sawkar and Thomas [1992], constructs a *compatibility graph* over nodes in $input(v)$ when v is being decomposed, in which two nodes are linked with an edge if they have a combined input size of K or smaller. A heuristic is used to repeatedly choose a pair of connected nodes according to a cost function, merge them, and update the edges connecting the merged node with other nodes. At the end of this procedure, each remaining node represents a bin. The cost function favors smaller combined input size and a small decrease of the number of edges in the compatibility graph. However, these extensions of bin-packing based decomposition apply only to leaf-DAGs.

3.1.4 *Structural Gate Decomposition for General Networks.* The recent work in Cong and Hwang [1996] investigates the problem of structural gate decomposition of *general unbounded networks* for depth-optimal LUT mapping. First, the authors showed that any further decomposition of gates in a K-mappable network leads to a smaller or equal mapping depth regardless of the decomposition algorithm used. That is, one should always decompose a network into a two-input network in order to obtain the minimum mapping depth. Therefore they focused on the following problem of structural gate decomposition for depth optimal K-LUT mapping (*the SGD/K problem*): Given an unbounded network N, decompose N into a two-input network N_2 such that for any other two-input network decomposition N_2' of N, the optimal LUT mapping depth of N_2 is less than or equal to the optimal LUT mapping depth of N_2'. When the input networks are restricted to only K-bounded networks (instead of unbounded networks), the resulting problem is called *the* K-SGD/K problem, which is a special case of the SGD/K problem. It was shown in Cong and Hwang [1996] that the SGD/K problem is NP-hard for $K \geq 3$ and that the K-SGD/K problem is NP-hard for $K \geq 5$. They developed an efficient heuristic algorithm for the K-SGD problem that combines the level-driven bin-packing decomposition technique for depth-optimal covering of trees (presented in the preceding subsection) and the network flow based node labeling technique for depth-optimal LUT mapping of K-mappable networks (presented in Section 4.3.1). This algorithm shows significant improvement in terms of both the final mapping depth and LUT count over the simple method of partitioning a

general unbounded network into trees and then decomposing each tree for depth-optimal mapping.

3.2 Symbolic Decomposition Methods

When the node is associated with a complex-gate function, node decomposition must consider the node functionality. A relatively simple approach is to manipulate the given representation of the function *symbolically*. For example, if the function is expressed as a cube cover (equivalently, an SOP), it can be treated as a polynomial and decomposed by polynomial factorization. Similarly, if the function is represented by an OBDD, decomposition can be performed by subgraph splitting. We regard such operations as *symbolic decompositions* in general. The complexities of these operations are usually functions of the representation size, which in the worst case is exponential to the input size of the gate.

3.2.1 AND-OR *Decomposition and Cube-Packing.* A node with SOP representation has a direct decomposition of a set of AND gates, each implementing a product term, and an OR gate with the AND gates as its inputs. This is called the *AND-OR decomposition*. Minimizing the SOP expression using two-level logic synthesis [Brayton et al. 1984] can result in a good and fast AND-OR decomposition, as the complexity is clearly linear to the number of literals. If a resultant node is not K-feasible, a simple-gate decomposition method will be applied. In the tech_decomp command of MIS-II [Brayton et al. 1987], balanced tree decomposition is used. A bin-packing based decomposition with consideration of input sharing, called *cube-packing*, is used in Murgai et al. [1991b]. In Weinmann and Rosenstiel [1994] the decomposition of the OR gate is viewed as the partitioning of the AND gates, and the Kernighan and Lin partitioning method [Kernighan and Lin 1970] is adapted.

3.2.2 *Algebraic Division Based Extraction.* An important type of symbolic operation on cube cover representation is *algebraic division*, which algebraically reexpresses a function f in the form of $f = p \cdot q + r$, where the variable in p (the *divisor*) and q (the *quotient*) are disjoint, and function r (the *remainder*) has as few cubes as possible. For example, $f = a \cdot c + b \cdot c + d = (a + b) \cdot c + d$ is a division with $p = (a + b)$, $q = c$ and $r = d$. If $r = 0$ the division is *even*. The important task for algebraic division is to choose a good divisor. The quotient of f divided by a cube is called a *primary divisor*, and if a primary divisor is *cube-free* (i.e., cannot be divided evenly by any cube), it is called a *kernel*, and the cubes that can be used to divide f to obtain the kernel are called its *cokernels*. In the previous example, $(a + b)$ is a kernel with cokernel c. Kernels and cubes (cokernels) are considered good candidates for divisors, and can be generated systematically [Brayton et al. 1990; Devadas et al. 1994].

When the node function is represented in SOP or any other literal based form, algebraic division can be used to identify subfunctions and create new nodes for the decomposition, which we call the *extraction based decomposi-*

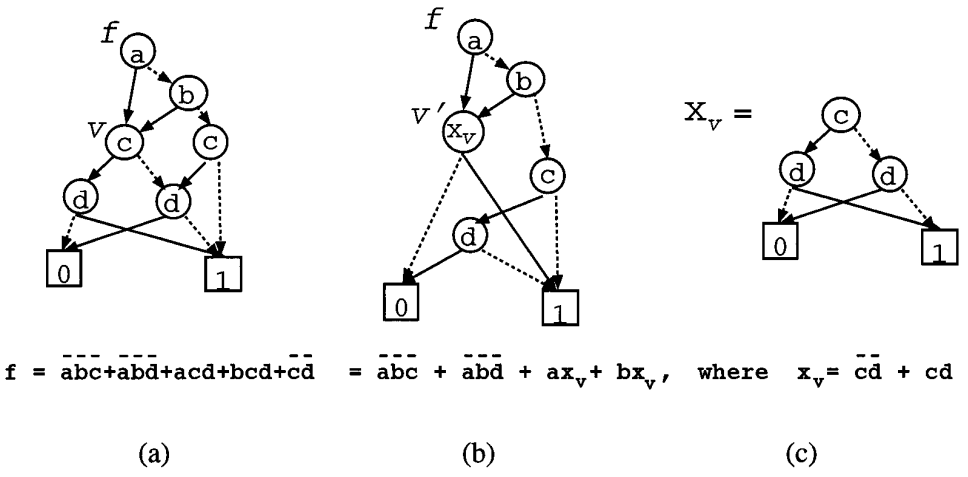

Fig. 5. Example of OBDD-based extraction.

tion, or *extraction* for short. Common subfunctions can be extracted from multiple nodes and shared in the decompositions of these nodes.

When there are multiple subfunctions to choose for extraction, a cost function can be evaluated for each of them, and the best one is chosen. In Murgai et al. [1991b], the cost is determined by an estimation of the number of LUTs needed to cover the network before and after the extraction, using the cube-packing heuristic. A simplified measurement used in Lu et al. [1994], is based on the change of the input numbers on the related nodes before and after the extraction.

One may order the variables before or during extraction, so that the kernels or cokernels of the variables with higher priority are extracted first. One application is to extract nodes with smaller levels first for depth minimization. In the *lexicographical factorization* approach [Saucier et al. 1993], a (partial or full) variable ordering is maintained by a *precedence matrix*, and *lexicalgraphical compatible* kernels (whose variables appear later in the ordering than those of their cokernels) are computed for extraction. As shown in Saucier et al. [1993b], this improves the routability of the synthesized network, and can also minimize area and/or depth by finding a good ordering.

3.2.3 OBDD *Based Extraction.* If the node function f is represented by OBDD, let v be a nonterminal with associated subfunction f_v. The subOBDD rooted at v defines f_v. Using the method of Chang and Marek-Sadowska [1992], f_v is extracted from f by replacing the subOBDD rooted at v with a single nonterminal v' such that $hi(v')$ is terminal 1, $low(v')$ is terminal 0, and $var(v') = x_v$ is a new variable representing the extracted function f_v in the input set of f. Figure 5 shows an example of this OBDD based extraction, which has several advantages. First, it is very simple and efficient, and the decomposed functions are instantly available in OBDD form. Second, common subfunctions are easily identified and extracted in a

multiroot OBDD, or among subOBDDs of a single-root OBDD. Moreover, because x_v is not an independent variable, it can be used to simplify other portions of the OBDD (Section 3.5.2).

The OBDD based extraction was used in a heuristic for global optimization of multilevel extractions in Stanion and Sechen [1995]. It recursively performs extractions until all functions are K-feasible. At each step it simultaneously performs a set of extractions, with the objective of minimizing the total *cost* of the resulting subfunctions. The cost of a function is an estimation of its size when decomposed into K-feasible functions. The implementation in Stanion and Sechen [1995] uses implicit enumeration to generate the sets of OBDD vertices to be extracted. Although it still cannot guarantee a decomposition of a minimum number of K-feasible subfunctions, this heuristic provides a way of quickly examining many extraction alternatives.

3.3 Boolean Decomposition Methods

The solution space of a symbolic decomposition method is limited by the given function representation. This lowers the computational complexity but also limits the effectiveness. Boolean decomposition methods go beyond the given representation and exploit the full functionality. For example, the *Boolean division* can be defined similarly to algebraic division in terms of $f = p \cdot q + r$, whereas $p \cdot q + r$ is a Boolean equivalence of f rather than a symbolic reexpression. Several heuristic methods for Boolean division can be found in Brayton et al. [1990]. In this subsection, we discuss two Boolean decomposition methods, *cofactoring* and *functional decomposition*, which are widely used for LUT logic synthesis.

3.3.1 *Cofactoring and Decision Diagram Conversion.* Shannon expansion $f = x \cdot f_{[x]} + \bar{x} \cdot f_{[\bar{x}]}$ defines a way of decomposition called *cofactoring*, where $f_{[x]}$ and $f_{[\bar{x}]}$ denote functions $f|_{x=1}$ and $f|_{x=0}$, and are called the *cofactors* of f with respect to x and $\bar{x}$, respectively. Because a cofactor has one fewer variable, a K-feasible decomposition on any node is possible by recursive application of cofactoring.

There are several ways to generalize cofactoring. First, other expansion forms such as the XOR based *Davio expansions* (see Becker and Drechsler [1995]) can be used similarly. Second, cofactoring with respect to a cube can be obtained by cofactoring with respect to each literal one by one. Moreover, cofactoring can be defined with respect to a function g as $f = g \cdot f_{[g]} + \bar{g} \cdot f_{[\bar{g}]}$ where f_g and $f_{\bar{g}}$ are functions satisfying the equation. Proper choice of g can result in simpler subfunctions with fewer variables. Finally, the Shannon expansion formula may be simplified to the form $f = x \cdot g_1 + \bar{x} \cdot g_2 + g_3$ where $f_{[x]} = g_1 + g_3$ and $f_{[\bar{x}]} = g_2 + g_3$, and g_3 are independent of variable x. The time complexity of cofactoring is linear to the size of the cube-cover representation.

Cofactoring based decompositions can be viewed as the conversion of decision diagrams to multilevel networks. For example, a complex gate can be decomposed into a network of MUX2 gates by converting its OBDD or

ITE representation into a network of MUX2 gates. The Davio expansions, which are based on XOR operation, can be used to obtain a network of AND and XOR gates. Although any complex gate can be decomposed into an AND/OR-network, a MUX2-network, or an AND/XOR-network using cofactoring, each form is suitable for certain types of gate functions, and should be chosen properly to obtain a good decomposition. Various studies have been done on this issue (See, for example, Heap et al. [1992], Saul [1991], Schafer and Perkowski [1993], and Thakur et al. [1995]). The approach proposed in Thakur et al. [1995] adds a MUX recognition heuristic in a combined AND-OR and Huffman tree decomposition procedure. If a node is recognized as a MUX, it is decomposed into MUX2 gates using a MUX decomposition algorithm instead of AND and OR gates; otherwise the normal AND-OR and Huffman tree decompositions apply. This approach can reduce the decomposition size and depth if many nodes do carry MUX type functions.

3.3.2 *Functional Decomposition.* The advantage of cofactoring based decomposition is that a simple procedure applies to any node function, but it is limited to the use of only cofactors as subfunctions. Functional decomposition exploits the possible use of *arbitrary* subfunctions. It is particularly suitable for LUT technology, as the decomposed functions can be implemented using LUTs directly as long as they are K-feasible, whereas in cell library based technology, they have to match the library cells. In fact, although the concept of functional decomposition was developed very early [Ashenhurst 1957], it has not been used very successfully in conventional logic synthesis [Brayton et al. 1990].

We first introduce some related concepts. In general, a functional decomposition is of the form $f(x_1, \ldots, x_r) = g(y_1(x_1, \ldots, x_i), \ldots, y_m(x_1, \ldots, x_i), x_j, \ldots, x_r)$, where $i, j, m \leq r$ and $i \geq j - 1 > 0$. Intuitively, this is a procedure of *encoding* the first $j - 1$ variables using m new variables. Therefore we refer the functions $y_1, \ldots, y_m$ as the *encoding functions*, and g as the *base function*. If $i = j - 1$, it is called a *disjunctive decomposition*, in which variables $x_1, \ldots, x_i$ form the *bound set*, and $x_j, \ldots, x_r$ form the *free set*; otherwise it is a *nondisjunctive decomposition*. If $m = 1$, it is called a *simple* decomposition; otherwise it is a *complex* decomposition. If $m < j - 1$, that is, g has fewer variables than f, the decomposition is *nontrivial*; otherwise it is *trivial*. For the purpose of obtaining K-feasible nodes, nontrivial decomposition is usually of interest, as it guarantees convergence of the recursive decomposition.

Ashenhurst Decomposition. The first functional decomposition method, *Ashenhurst decomposition* [Ashenhurst 1957] solves the simple disjunctive decomposition problem by giving the sufficient and necessary condition. It uses a *partition matrix*, in the form of a 2-D truth table, for a given variable partitioning of bound set $B = \{x_1, \ldots, x_i\}$ and free set $F = \{x_{i+1}, \ldots, x_r\}$. Each column corresponds to one possible assignment (i.e., a minterm) of the bound set variables, and each row corresponds to one possible assignment of the free set variables. The partition matrix implies a simple

disjunctive decomposition if and only if there are at most two distinct columns (thus the bound set variables can be encoded into one bit, using one function). To have a nontrivial decomposition, the size of the bound set must be at least two. The derivation of the base and encoding functions is easy: the encoding function $y(B)$ can be defined by the bound set minterms corresponding to the columns of one pattern (which is equivalent to assigning $y = 1$ for these columns); the base function $g(y, F)$ can be defined by the compressed truth table obtained after merging the identical columns and assigning the value of y for each column.

Ashenhurst decomposition can be easily generalized to the case of complex disjunctive decomposition, as was first proposed in Curtis [1961]: if there are at most 2^m different columns, the bound set can be encoded by m bits using m encoding functions. To have a nontrivial decomposition, the size of the bound set must be larger than m. The encoding and base functions can be obtained similarly based on the encoding.

Extensions of Ashenhurst decomposition to multilevel disjunctive decompositions can be found in Ashenhurst [1957] and Curtis [1963]. Ashenhurst decomposition has also been generalized to incompletely specified functions. In Wan and Perkowski [1992], a graph is constructed with the columns as vertices, and two columns are connected with an edge if they cannot be made identical by properly specifying don't-care entries. The problem is then reduced to the *graph coloring* problem and solved by a greedy heuristic.[7]

A partition matrix defined by the chosen bound and free sets may not yield any nontrivial decomposition. Such a "bad" partition matrix may be converted into a "good" one using a patching method proposed in Wan and Perkowski [1992]. Specifically, if the partition matrix P of function f can be decomposed into two matrices P_1 and P_2, such that for each corresponding entry (i, j), $P(i, j) = P_1(i, j) \oplus P_2(i, j)$, and P_1 and P_2 define decomposable functions f_1 and f_2, respectively, then $f = f_1 \oplus f_2$, and a decomposition can be obtained using the current bound/free set partitioning by decomposing f_1 and f_2 separately.

Roth-Karp Decomposition. A major drawback of the Ashenhurst decompositions is that they are based on the partition matrix (also called *decomposition chart* in Curtis [1961, 1963]), which is a 2-D truth table, thus the time and space complexity is always $O(2^r)$ for an r-variable function. If r is large, this method is impractical due to the high memory requirement. A more efficient way of solving complex disjunctive decomposition is the *Roth-Karp decomposition* [Roth and Karp 1962], which also considers nondisjunctive decomposition. It works on the ON and OFF sets of f in the form of cube covers, which can be substantially smaller in size than the truth table.

[7] The graph-coloring problem is to use the minimum number of colors to color the vertices, such that two vertices do not have the same color if they are connected by an edge.

Given a cube cover representation of a function and a partitioning of variables into bound and free sets, each cube c can be partitioned into a bound-part c_b and free-part c_f, that is, $c = c_b c_f$. An *incompatibility graph* is constructed with the minterms of the bound set variables as the vertices. The edges are determined as follows. For each cube c in the ON set and each cube d in the OFF set, if c_f intersects d_f (i.e., they cover a common minterm), then c_b and d_b are said to be *incompatible*, and an edge (v, w) is added between each vertex v corresponding to a minterm covered by c_b and each vertex w corresponding to a minterm covered by d_b. (In terms of the partition matrix, this implies that there is at least one row, corresponding to a common minterm of c_f and d_f, on which the entries at the columns corresponding to c_b and d_b are different.) After the incompatibility graph is constructed, the problem is reduced to the graph-coloring problem, and a set of vertices with the same color are mutually compatible (corresponding to a collection of identical columns in the partition matrix). Consequently, an encoding of m bits exists if and only if the incompatibility graph can be colored using no more than 2^m colors. Derivation of the encoding and base functions is similar to the case of Ashenhurst decomposition.

In Roth and Karp [1962], nondisjunctive decomposition was also studied, and two alternatives were proposed. One is to treat the appearance of bound set variables in the free set as single variable encoding functions generated by the decomposition. This restricts the flexibility of encoding, but if the nondisjunctive decomposition exists, a valid encoding will be found. The other approach is to replicate the shared variables in the cube representation to make the two sets disjoint. When the literal of a shared variable x does not appear in a cube c, however, c must be split into two, one containing x and the other containing $\bar{x}$, before the replication.

OBDD Based Decomposition. Although Roth-Karp decomposition uses cube cover representation, which is more compact compared with the partition matrix of Ashenhurst decomposition, for complex functions the representation may still be large. Moreover, because the graph-coloring problem is NP-hard [Garey and Johnson 1979], Roth-Karp decomposition has high time complexity: for a bound set of k variables, the number of vertices in the incompatibility graph is 2^k, and the worst case complexity of solving the graph-coloring problem is $O(2^{2^k})$.

If the function is represented in OBDD (which is often smaller than cube cover), functional decomposition can be done directly on OBDD in a very simple way. (In recent years, this approach has been used in Chang and Marek-Sadowska [1992], Cong and Ding [1993b], Lai et al. [1993a]).

Given a partitioning of the variable set X into bound set B and free set F, an OBDD is built such that the variables in B appear before the variables in F. Then the vertices are partitioned into two sets, U and L, where U contains the vertices labeled with the variables in B, and L contains the vertices labeled with the variables in F and the two terminals. (If the OBDD is drawn with the root at the top and vertices aligned for each variable, such partitioning corresponds to a horizontal line drawn immedi-

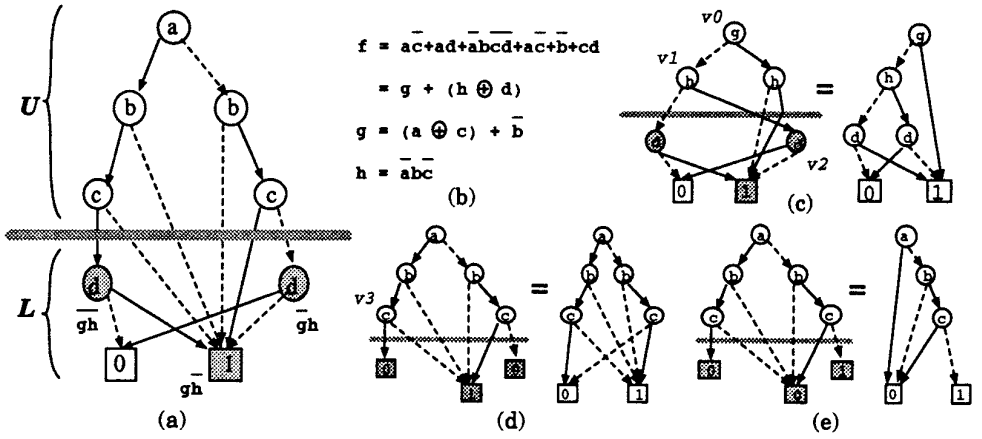

Fig. 6. OBDD based decomposition: (a) original OBDD, with bound set $\{a, b, c\}$, free set $\{d\}$, and minterm assignment of nodes in C_B; (b) corresponding function and its decomposition; (c) construction of base function; (d, e) construction of encoding functions.

ately below the $|B|$th level.) This partitioning defines a *cut* of the OBDD which is the set of vertices in L (denoted C_B) that have incoming edges from vertices in U. The size of C_B, denoted s_B, is called the *size* of the partitioning. Figure 6(a) shows an example of a cut in an OBDD.

If $s_B \leq 2^m$, it means that the first $|B|$ variables induce no more than 2^m distinct functional states, and thus can be encoded in no more than m bits, or with m encoding functions in a decomposition. [In Figure 6(a), the three shadowed nodes are in C_B, so $s_B = 3$ and a 2-bit encoding is needed.] This condition is easy to check, and the base function and the encoding functions can be obtained efficiently in OBDD form from the original OBDD as follows.

Assuming that $2^{m-1} < s_B \leq 2^m$, then there will be m encoding functions $y_1, \ldots, y_m$. We assign the first s_B minterms of new variables $y_1, \ldots, y_m$ to the vertices in C_B. To construct the base function, the U part of the OBDD is first replaced by a full OBDD of the m variables $y_1, \ldots, y_m$ in which each of the 2^m terminals corresponds to a minterm of $y_1, \ldots, y_m$. [In Figure 6(c), the upper portion of the OBDD associated with variables a, b, and c is replaced by a full BDD of g and h first.] Next we replace each terminal with a vertex in C_B that was assigned with the same minterm if it exists, or an *arbitrary* cut vertex otherwise—a don't-care case. [In Figure 6(c), path $\langle v_0, v_1, v_2 \rangle$ corresponds to minterm $\bar{g}h$, which was associated with v_2.] Then we reduce the resulting OBDD which represents the base function. To construct the encoding function y_i, the L part of the OBDD is modified by removing the noncut vertices and replacing each cut vertex v by terminal 0 if $\bar{y}_i$ appears in the minterm associated with v, or by terminal 1 if y_i appears in the minterm. Finally, we reduce the resulting OBDD, which yields an OBDD representation of encoding function y_i. [In Figure 6(d), encoding function g is constructed. For example, we have $hi(v_3) = 0$, because originally $hi(v_3)$ was associated with minterm $\bar{g}\bar{h}$.]

This OBDD based functional decomposition has a clear advantage over both Ashenhurst and Roth-Karp decompositions. First, the OBDD representation is compact, and conversion from or to other forms of representation is simple. Second, all operations are either linear to the size of the OBDD, or exponential only in terms of the size of bound set (which is usually K or a smaller constant in LUT network synthesis). Moreover, it is easily extended to the case of nondisjunctive decomposition by allowing the partitioning line to cross the levels, that is, by allowing some vertices labeled with variables in B to be partitioned into the L part [Lai et al. 1993a]. Because such cross-level partitioning may not be unique, there are usually multiple choices for nondisjunctive decompositions, resulting in base and encoding functions of different sizes and structures.

Multi-Output Functional Decomposition. If multiple node functions are considered for simultaneous decomposition, as in the procedure of a recursive multilevel decomposition, it is beneficial to find common subfunctions. This problem was first studied in Karp [1963], where the condition for a shared subfunction of two functions f_1 and f_2 was presented. Note that Roth-Karp decomposition defines an *equivalence relation* over the bound set minterms, where each equivalence class consists of the minterms of the same code under the encoding. Moreover, each encoding function defines a *valid partitioning* that partitions the equivalence classes into two sets, each containing no more than 2^{m-1} classes. Therefore a shared encoding function of two node functions should define a valid partitioning for both functions. The method used in Karp [1963], applicable only to two-output decomposition, enumerates the possible partitionings to find such shared encoding functions.

The approach of He and Torkelson [1993] and Wan and Perkowski [1992] extends Ashenhurst decomposition for multiple functions $f_1, f_2, \ldots, f_r$. Given the bound set and free set and a fixed variable order, let P_i be the partition matrix for f_i. A new partition matrix P is constructed by superimposing all partition matrices P_is such that each entry in $P(i, j)$ is a bit vector $\langle P_1(i, j), \ldots, P_r(i, j)\rangle$. To find *common* encoding for *all* functions, P is used. It is easily seen that the existence conditions of Ashenhurst decomposition are still valid, but are much more difficult to satisfy because each entry has 2^r possible values instead of 2. This approach is more useful for incompletely specified functions, as proper assignment of the don't-cares can help to create identical columns in P. An OBDD version of this approach was implemented in Lai et al. [1994].

A more general multi-output decomposition method proposed in Lai et al. [1994] finds *shared* encoding among *some* output functions. Specifically, its objective is to find a minimum set G of encoding functions for the distinct columns of *all* partition matrices $P_1, \ldots, P_r$, such that for each P_i, there is a subset $G_i \subseteq G$ of encoding functions that encode the distinct columns of P_i, and $|G_i|$ is less than the bound set size. Clearly, such an encoding induces a nontrivial decomposition for each output function, and some encoding functions can be shared among the output functions. In Lai et al.

[1994] a heuristic was used to repeatedly select the "most shared" encoding functions until all output functions were properly encoded. The decomposition was implemented using OBDD representation.

A limitation of this method is that its encoding is over the *distinct* columns. Therefore identical columns in different partition matrices cannot be assigned different codes. This is called a *strict-* or *uni-coding*. On the other hand, a *nonstrict-* or *multi-coding* can assign different codes to such columns. Strict-codings are easier to compute than nonstrict-codings, but they may not result in the optimal decomposition.

The two-output decomposition in Karp [1963] is able to use nonstrict-coding inasmuch as its encoding is over the bound set minterms of each function, which are equivalent to its partition matrix columns. Nonstrict-coding based multi-output functional decomposition was recently addressed in Wurth et al. [1995]. It formulates the general decomposition and encoding problems as the *refinement* of equivalence relations on the minterms of the bound set variables, where one equivalence relation R_i is *refined* by another R_2 if each class of R_2 is contained in a class of R_1. A valid encoding G defines an equivalence relation R_G that refines the equivalence relation R_P defined by the partition matrix P on its columns. For multi-output decomposition where the partition matrix for output f_i is P_i, R_G must refine the *product* of all the relations R_{P_i}, where the product is defined as the minimum common refinement. Moreover, for nontrivial decomposition at every output f_i, there must also be a partial encoding $G_i \subseteq G$ of size less than the bound set size, which defines an equivalence relation R_{G_i} that refines R_{P_i}. Such encoding is over the equivalence classes of the product relation, whose encoding space is between the number of all columns and the number of all distinct columns. Therefore nonstrict-coding is used if necessary. For efficient implementation, OBDDs were used to represent the functions, their equivalence relations, the product of these relations, as well as the characteristic function used in the (reduced) implicit enumeration of the encoding functions [Wurth et al. 1995].

Base and Encoding Function Optimization. Given a variable partitioning, there are often many ways to construct the base function and encoding functions that leave room for optimization. Because the base function will be further decomposed in many cases, it is beneficial to make its structure simple. This was discussed in Murgai et al. [1994]. Also, the algorithms in Sawada et al. [1995] exploit the don't care set to reduce the number of variables for the base function.

If the encoding functions are also subject to further decomposition, their structural simplicity is desired as well. In Chang and Marek-Sadowska [1992], a heuristic was presented to reduce the total number of vertices in the OBDDs of the encoding functions. Even if the encoding functions are not to be further decomposed, it is still beneficial to make them simple. Several recent works [Cong and Hwang 1995b; Huang et al. 1995; Legl 1996] have focused on the *support minimization* of encoding functions, where the objective is to make as many encoding functions as possible

independent of one or more bound set variables. Such encoding functions result in LUTs of smaller input sizes, which are more easily merged with other LUTs, or packed into programmable logic blocks containing multiple LUTs with limited number of total inputs. Such LUTs may also have better routability due to the reduced number of connections. The algorithm in Huang et al. [1995] encodes equivalence classes according to the set of *dichotomy* to eliminate variables from the bound set. It was designed for packing encoding functions into Xilinx XC3000 programmable logic blocks, which can implement either a 5-LUT or two 4-LUTs with no more than five inputs in total [Xilinx 1994]. However, it only considers strict-coding. The method in Legl et al. [1996] is able to compute nonstrict decomposition for support minimization. It uses an implicit algorithm to consider all possible (constructable) encoding functions. The result in Cong and Hwang [1995b] gives a necessary and sufficient condition for support minimization of the encoding function, which can be used to test whether there exists a functional decomposition for a given variable partitioning such that some encoding functions could depend partially on variables in the bound set. Based on this condition, an algorithm is developed for computing a maximal set of encoding functions that depend partially on variables in the bound set. This algorithm can perform nonstrict-coding, and can also be used for nondisjunctive decomposition and multiple-output decomposition.

Variable Partitioning. All the functional decomposition techniques presented so far assume a given partitioning of variables into the bound set and free set. An open problem is how to choose the proper variable partitioning. It involves two issues—how to choose the best partitioning for the *current* decomposition, and how to choose the best partitioning for the *overall* decomposition process in the case of recursive decomposition. The latter is obviously a more difficult problem and has not been studied. As for the former, several approaches have been suggested. The quality of a decomposition is usually measured by the number of variables of the base function after decomposition, or by the depth of the decomposition. In Lai et al. [1993a], an enumeration method was proposed using an extended OBDD representation called *EVBDD* [Lai and Sastry 1992]. Because the number of different partitionings is exponential, exhaustive enumeration is often impractical. In Cong and Ding [1993b], the inputs (variables) are ordered in increasing order of delay, so that inputs with smaller delays are first chosen as bound set variables. An iterative improvement heuristic was proposed by Hwang et al. [1992], which adopts the Kernighan-Lin partitioning method to partition the variables into bound and free sets using a heuristic cost function. Recently, in Shen et al. [1995], a heuristic method was described to greedily choose the "most compatible" variable for the bound set one by one.

3.4 Upper Bounds on Node Decomposition

Because of the extremely large solution space and the interdependency of logic optimization and technology mapping, it is difficult to quantitatively

measure the optimality of logic optimization for LUT logic synthesis, even in the case of the decomposition of a single node. However, based on the decomposition techniques discussed in this section several upper bounds can be derived in terms of the number of LUTs needed to cover the decomposed node.

First, the exact (lower and upper) bound for a node with an m-input simple-gate function is $\lceil (m - 1)/(K - 1) \rceil$ as we have shown in Section 3.1.1.

For a node with complex function in the sum-of-product representation, if it has c product terms, an upper bound is $\lfloor (m - 1 + (c + 1)(K - 2))/(K - 1) \rfloor$ when $K > 2$ [Murgai et al. 1993a]. In general, for a node with complex function, an upper bound based on Shannon expansion is $2^{m-K+1} - 1$ for $K > 2$, because each expansion can be implemented by a MUX2 (thus an LUT). For larger values of K, some of the MUX2 gates in the Shannon expansion tree can be merged to give a better bound. For example, in Murgai et al. [1991b], an upper bound of $2^{m-4} - 1 - (2^{m-5} - x)/3$ was derived for $K = 5$, where $x = 1$ for odd m and $x = 2$ for even m. For $K = 6$, each LUT can implement two levels of Shannon expansion, resulting in a bound roughly one third of the MUX2 count in Shannon expansion.

When $m = K + c$ for some (small) constant c, a constant number of K-LUTs will suffice (e.g., via Shannon expansion). For instance, three K-LUTs are sufficient for $m = K + 1$ when $K > 2$ (four for $K = 2$). In Murgai et al. [1993a] this was shown to be a lower bound as well.

3.5 Network Simplification Techniques

The *network simplification* operations are carried out among a group of nodes based on the fact that each node v can be associated with different functions using different input variable sets. For example, if $input(v) = \{s, t, u\}$ and $f_v = f_s + f_t \cdot f_u$, and there is another node w with $input(w) = \{s, t\}$ and $f_w = f_s + f_t$, then f_v can be reexpressed as $f_v = f_w \cdot (f_s + f_u)$, with $input(v) = \{s, u, w\}$. This is called *substitution*, as f_w is used to substitute an expression in f_v. Similarly, if $f_v = f_w \cdot (f_s + f_u)$ is the original function associated with v, and $f_v = f_s + f_t \cdot f_u$ is more desirable, the reversed transformation will also be a simplification operation, called *collapsing*, as w is eliminated from $input(v)$ and "collapsed" into v. As another example of collapsing, still assume that $input(v) = \{s, t, u\}$ and $f_v = f_s + f_t \cdot f_u$. In addition, assume $input(u) = \{s, t\}$, and $f_u = \overline{f_s} \cdot \overline{f_t}$. By collapsing u into v, we can get $f_v = f_s$, a *support reduction* for v from three inputs to one (in fact, v can be eliminated and its output supplied by s).

Another important source of freedom for network simplification is the use of don't-cares. For a multilevel network, don't-cares on primary inputs can be specified explicitly for each primary output, and are called *external don't-cares*. Don't-cares can also exist implicitly on the inputs of any node due to the *structural redundancy* of the network, and are called *internal don't-cares*. In the previous example where $f_v = f_s + f_t \cdot f_u$, and $f_u = \overline{f_s} \cdot \overline{f_t}$,

the input vectors {011, 101, 111} for $input(v) = \{s, t, u\}$ are not possible, thus they belong to the DC_{f_v}, and can be used to simplify f_v. These are called the *satisfiability don't-cares* as they represent input assignments impossible to satisfy. The other type of internal don't-cares are the *observability don't-cares*. When an input vector is applied to the inputs of a node, the output of the node may be *blocked* by other signals and cannot be observed at the primary output. For example, if node v with function $f_v = f_s + f_t \cdot f_u$ has a single fanout w which with $input(w) = \{u, v\}$ and $f_w = f_u + f_v$, then the output of v on input vectors $xx1$, where x can be 0 or 1, will not affect the output of w which remains 1. Therefore these vectors also belong to DC_{f_v}, and can be used to simplify f_v into $f_v = f_s + f_u$ for support reduction.

In this section we introduce several network simplification techniques that have been used for LUT logic synthesis, but will leave out many of the details, which can be found in the cited literature. Most methods for simplification in conventional logic synthesis are also useful in LUT logic synthesis, and can be found in Brayton et al. [1990], Devadas et al. [1994], and De Micheli [1994].

3.5.1 *Local Simplification.* A commonly used technique in logic optimization is to collapse a subnetwork and redecompose it for a certain purpose. In Murgai et al. [1991b] an operation called *move-the-fanins* was used. For an infeasible node v, each of its inputs $w \in input(v)$ is examined. If w is fanout-free, the operation will try to collapse w into v and redecompose v into v' and w' such that both are K-feasible. If this operation succeeds, some fanins of v are effectively "moved" to w. A similar operation called *gate decomposition* was used in Cong et al. [1992a]. It looks for a subset of two or more nonPI fanins $u_1, \ldots, u_m$ of v whose combined input size is bounded by K. If such a set exists, the operation tries to decompose v into v' and v'', such that v' is a fanin of v'' and carries the fanins $u_1, \ldots, u_m$, while v'' carries the remaining fanins of v. If this is possible, then $u_1, \ldots u_m$ can be collapsed into v'. As a result v'' replaces v with fewer number of fanins and the total number of nodes reduced by $m - 1$. This can be viewed as a simple disjunctive decomposition with restricted bound set selection followed by a collapsing. This operation has also been generalized to the case where the fanins $u_1, \ldots, u_m$ are not fanout-free, but the decomposition and collapse are simultaneously applicable to all fanouts.

For both operations, there can be multiple choices of the fanin and root nodes. In Cong et al. [1992a], candidates for the operations were represented as the edges of a (hyper)graph over the gates, and a *maximum matching* was computed in $O(n^3)$ time [Gabow 1976] (where n is the number of gates in the network) to decide how to apply a set of gate decomposition operations simultaneously without conflict.

3.5.2 *Rule-Based Global Simplification.* Local simplifications are relatively efficient, but may not be very powerful due to the range limit. One way of maintaining the efficiency while overcoming the range limit is to use a set of transformation *rules* instead of exhaustive search in more global simplification operations. Here we show two examples.

In the approach of Chang and Marek-Sadowska [1992], when an OBDD-based extraction is performed on a function f, a set of rules is used to simplify $OBDD_f$. During the extraction of f_v at nonterminal v, a new variable x_v is introduced to represent the extracted subfunction f_v. A set of rules is used to identify subOBDDs of $OBDD_f$ that have similar structure to the extracted subOBDD $OBDD_{f_v}$, and restructure them so that part of them will be the same as $OBDD_{f_v}$, and thus can be replaced by x_v. For example, if nonterminal vertex w in $OBDD_f$ satisfies $var(v) = var(w)$ and $hi(v) =$ hi(w), then after f_v is extracted and replaced by the new variable x_v associated with a new nonterminal vertex v', we can simplify $OBDD_f$ by letting $hi(w) = v'$ and eliminating the subOBDD of $hi(w)$. This can be proved by substituting $f_{hi(w)}$ with $f_v{}'$ in the Shannon expansion of f_w, and applying the condition $hi(v) = hi(w)$. More rules of such simplification can be found in Chang and Marek-Sadowska [1992].

Another rule based simplification method shows interesting use of internal don't-cares. In Chang et al. [1994], the objective is to reduce the interconnection congestion by patching the network, in particular, by replacing one or more connections with another set of connections (and possibly new gates). To remove a congested connection (*target wire*), a set of rules is derived to introduce new connections (*alternate wires*) in a less congested area, which will not alter the functionality at the primary outputs, but will block the observability of the target wire, so that it becomes redundant and can be removed.

3.5.3 *Permissible Function Based Simplification.* For a node v, consider the function f_{MC_v}, that is, its function in terms of the primary inputs. The existence of don't-cares, in particular, observability don't-cares, makes f_{MC_v} an incompletely specified function, which therefore can be implemented in different ways (by covering different portions of the DC set). These different functions are called *permissible functions*, and the *maximum set of permissible functions* (MSPF) covers all possible implementations of the node function. In Muroga et al. [1989], a procedure to compute MSPFs of all the nodes, as well as a more efficient procedure to compute a subset of MSPF for each node, called a *compatible set of permissible functions* (CSPF), was developed. The latter computes the CSPF in one pass from the primary outputs to the primary inputs while identifying and removing redundant connections. (The original implementation was based on cube cover representation. An OBDD version was due to Matsunaga and Fujita [1989].) The knowledge of the permissible functions can be used to simplify the network, including the removal of redundant connections and nodes (if the connection or node has a trivial permissible function), substitution of connections and nodes (if one connection or node carries a permissible function of the other), and merging of nodes (if two nodes have a common permissible function), and so on. Substitution by existing connections or nodes results in elimination of substituted connections and/or nodes, whereas substitution by new connections and/or nodes results in replacement. These operations are powerful, but usually of high time and

space complexity. We outline several applications of the permissible function-based network simplification.

Input (or *support*) reduction is an important objective in network simplification for LUT logic synthesis. The *minimal dependence set* problem [Halatsis and Gaitanis 1978] on a node v and a set of nodes $S = \{u_1, \ldots, u_m\}$ that are not successors of v is to find a *minimal* subset $S^+ \subseteq S$ such that a permissible function of v can be expressed by a composition of permissible functions of the nodes $w \in S^+$. That is, the function of v can be reexpressed using S^+. A related problem is to find a good set of candidates that results in good minimal dependence sets. For the first problem, a cube cover-based method was proposed in Fujita and Matsunaga [1991], and an OBDD-based implementation was proposed in Chen [1992]. For the second problem, a set of conditions for legitimate candidates were identified in Chen et al. [1991] which can be used to choose candidate sets, as in Chen [1992].

Node reduction (elimination) is another important objective. A node can be eliminated when all its fanout nodes have their functions reexpressed using other nodes as inputs. Clearly, fanout-free nodes are the easiest candidates. In Chen [1992], the approach was to give fanout-free nodes a higher cost when computing minimal dependence sets, so that they are less likely to be used. A more global approach was taken in Chen and Cong [1992], where for each fanout-free node v, the condition of eliminating v is computed and a maximum set of simultaneously removable fanout-free nodes is identified and removed. However, this approach is restricted in that, to construct the replacement functions for the fanouts of the target node, observability don't-cares cannot be used because simultaneous removal of multiple nodes in a network will not preserve the observability conditions derived for each single node.

Another use of permissible function-based simplification was demonstrated in Kukimoto and Fujita [1992] where the problem of *patching* a network without changing the layout (i.e., interconnections) is considered. The algorithm selects a set of candidate nodes to change, derives a Boolean relation that captures the allowed permissible functions of the selected nodes for implementing the changed design, restricts the relation so that the input to each node is not changed, and then derives a permissible function for each selected node from the reduced relation. The candidates are chosen from nodes of the same level, so that no node is the transitive fanin/fanout of another node, which simplifies the derivation of the Boolean relation.

4. TECHNOLOGY MAPPING

After a multilevel network of generic gates is optimized using the techniques described in the preceding section, the technology mapping step transforms it into an LUT network by covering it with LUTs. In this section we present commonly used techniques for LUT technology mapping. We organize this section based on the types of the input networks—tree, leaf-DAG, MFFC, and general network. For each type of the networks, we

consider techniques for area minimization, depth minimization, and general delay minimization. Routability and power minimization are briefly discussed at the end of this section.

Although in a mapping solution only a subset of nodes is implemented by LUTs (the remaining nodes are covered by the LUTs implementing others), many mapping algorithms compute the LUT implementation for *every* node, and then choose a subset of nodes to implement. Therefore in such cases we simply illustrate the mapping technique in terms of how it maps a generic node v in the network.

4.1 Tree Mapping

Assume that the input network is a K-bounded tree T_r with root r. If LUT_r implements r in a mapping M_{T_r} of T_r, then M_{T_r} induces a mapping M_{T_w} for each subtree T_w rooted at a node $w \in input(LUT_r)$. Denote the number of LUTs in mapping M_{T_v} as $area(M_{T_v})$, and define $area(M_{T_v}) = 0$ if v is a PI. Then $area(M_{T_r}) = 1 + \Sigma_{w \in input(LUT_r)} area(M_{T_w})$. Therefore the area optimal mapping $M_{T_r}^*$ of T_r can be determined by the area optimal mappings of its subtrees; that is, $area(M_{T_r}^*) = \min_{LUT_r}\{1 + \Sigma_{w \in input(\mathrm{LUT}_r)} area(M_{T_w}^*)\}$. This recursive relation suggests the use of dynamic programming, which computes the area optimal mapping of each subtree T_v for $v \in T_r$ in the topological order starting from the PIs in T_r.

Given the area optimal mapping of each subtree T_v in T_r ($v \neq r$), a straightforward approach to find the best LUT_r is to enumerate all K-feasible cones rooted at r [Francis et al. 1990]. Because T_r is a tree, the number of such cones is a constant depending only on K [Cong and Ding 1994b]. However, it was shown in Farrahi and Sarrafzadeh [1994b] that a greedy algorithm will also produce an optimal solution for this problem. Let $input(v) = \{w_1, \ldots, w_m\}$, and the LUT implementing w_i in an optimal mapping of T_{w_i} be LUT_{w_i}. Moreover, without loss of generality we assume that $|input(LUT_{w_1})| \leq \cdots \leq |input(LUT_{w_m})|$. Then the greedy packing $LUT_v = \{v\} \cup_{i \leq s} LUT_{w_i}$, where s is the largest index such that LUT_v remains K-feasible, gives an area optimal mapping for T_v. Based on this property, at each node v, we sort the input LUTs of the nodes $w_i \in input(v)$ in increasing order of input size, greedily expand the cone C_v to cover as many input LUTs as possible in that order, and let $LUT_v = \mathrm{C}_v$. This algorithm, named *tree-map* in Farrahi and Sarrafzadeh [1994b], gives the optimal mapping solution of T_v and has time complexity of $O(\max\{K, \log n\} \cdot n)$ on a tree of n nodes. Figure 7 shows an example of *tree-map*.

A similar approach can be used to obtain a depth-optimal mapping by changing the sorting criterion—instead of the increasing order of the input size, we sort the input LUTs in decreasing order of their depths. This method can also be easily generalized to other delay models.

If the K-bounded tree is obtained using the bin-packing based decomposition described in Section 3.1.3, the *tree-map* algorithm will reproduce the bin-packing result. However, the bin-packing approach has the advantage

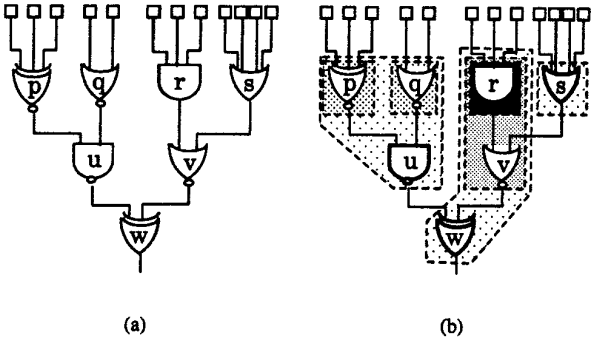

Fig. 7. Area optimal mapping for tree: (a) original network; (b) mapping for $K = 5$. Nodes u, s, w will be implemented. Input LUT sorting and partial packing happens at v and w.

of achieving decomposition and mapping simultaneously when the tree is not K-bounded [Francis et al. 1991a,b].

Tree mapping has limited use because real networks are rarely trees. Mapping for leaf-DAGs is more useful, inasmuch as a general network can be partitioned into a set of leaf-DAGs. For depth-optimal mapping (or delay-optimal mapping for any static delay model), the *tree-map* mapping procedure will still produce an optimal solution, as a shared fanin can simply be treated as different inputs. Allowing multifanout input to be treated as multiple independent inputs is a very important property of the static delay models, as it simplifies the optimization problem and allows very efficient mapping methods. For area minimization, the general dynamic programming approach still produces an optimal mapping solution, but the *tree-map* algorithm is no longer optimal. If all K-feasible cones of v are to be checked as candidates for LUT_v, the number is not bounded by a constant, but a polynomial of n (for fixed K). Enumeration methods are introduced in the next subsection.

4.2 MFFC Mapping and Duplication-Free Mapping

Mapping of an MFFC is more complicated than that of a leaf-DAG because internal nodes may also have multiple fanouts, and the LUTs may overlap. In fact, general mapping in an MFFC is as difficult as general mapping in an arbitrary network. Because $area(M_{MFFC_v}) = 1 + area(M_{MFFC_v - LUT_v})$, an optimal mapping of $MFFC_v$ implies an optimal mapping of $MFFC_v - LUT_v$, which may be any general network.

If we restrict the mapping solution to be *duplication-free*, that is, each node is covered by only one LUT, the problem is simplified, both for an MFFC and also for a general network. In particular, it was shown in Cong and Ding [1993a] that in any duplication-free mapping, LUT_v is always *contained* in $MFFC_v$. This implies that (1) to find the best LUT_v in a duplication-free mapping, we only need to search the K-feasible FFCs inside $MFFC_v$; and (2) for area minimization, duplication-free mapping of a general network can be performed optimally by partitioning the network

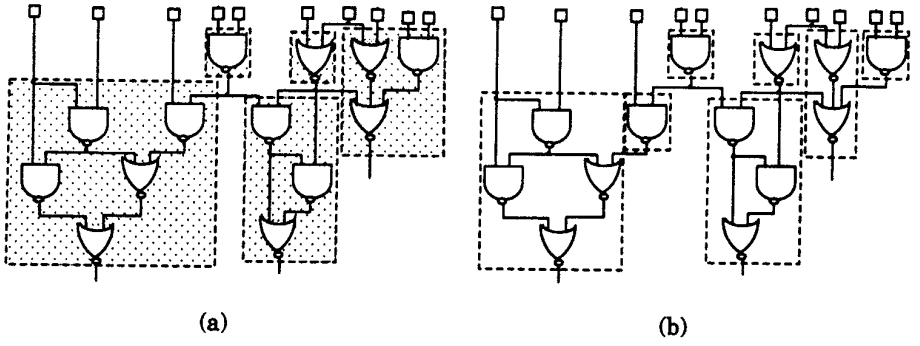

Fig. 8. Example of duplication-free mapping: (a) original network and MFFC partitioning; (b) duplication-free mapping for $K = 3$. Note that LUTs do not go across MFFCs.

into a set of MFFCs and computing the optimal mapping of each MFFC independently.

These results suggest a dynamic programming approach for duplication-free mapping. Given a network, for each node v in topological order, an area optimal mapping of $MFFC_v$ is computed. When we map $MFFC_v$, for each node $w \in MFFC_v$ other than v, an optimal mapping of $MFFC_w$ is already known. The optimal mapping of $MFFC_v$ consists of the proper selection of LUT_v, the LUT for v, and an optimal mapping of $MFFC_v - LUT_v$, which can in turn be partitioned into disjoint MFFCs, whose optimal mappings have been computed already, thus $area(MFFC_v - LUT_v)$ can be calculated easily. Therefore an optimal mapping of $MFFC_v$ can be obtained by finding the best LUT_v that minimizes $area(MFFC_v - LUT_v)$. This method was called *df-map* in Cong and Ding [1993a]. An example is shown in Figure 8.

In order to find the best LUT_v, all K-feasible FFCs of v are enumerated, which is achieved by enumerating all *K-feasible cuts* of $MFFC_v$. In Cong and Ding [1993a], the enumeration of all K-feasible cuts was done using tree-based recursion. A spanning tree ST_v of $MFFC_v$ was used and all K-feasible cuts in ST_v were generated. These cuts were then modified to include the starting nodes of the edges not in ST_v (called *escape nodes*), when necessary. It was shown that the total number of K-feasible cuts in $MFFC_v$ is bounded by a polynomial of $|MFFC_v|$ for a given constant K. It was also shown in Cong and Ding [1993a] that there exists an area optimal mapping solution, in which each K-feasible MFFC is contained in an LUT. This leads to a preprocessing step that collapses all K-feasible MFFCs, which often reduces the number of nodes in the network significantly.

Duplication-free mapping for area minimization is of interest for the following reasons. First, it is a good approximation of general mapping for area minimization, as logic duplication increases the number of gates and the interconnection density, which may result in more LUTs and/or a large routing area in the mapping solution implementation. Second, area optimal mapping with logic duplication is difficult (in fact, NP-hard, as shown in the next subsection). Nevertheless, proper logic duplication can be benefi-

cial to area minimization in some cases. For example, if a large number of unsaturated LUTs exist after duplication-free mapping, proper logic duplication can take advantage of the extra LUT capacities and reduce the LUT count.

Although *df-map* can be easily modified for depth-optimal duplication-free mapping, there is no such need—first, delay-optimal mapping often uses a considerable amount of logic duplication to increase the parallelism in the network; second, there exist more efficient algorithms for depth-optimal mapping of general networks.

4.3 General Network Mapping

One approach to general network mapping is to partition the network into a set of MTs or MFFCs and map each of them separately. Tree partitioning has been a common practice in cell library based technology mapping [Keutzer 1987]. Such approaches, however, often compromise the mapping quality. In this section, we present direct mapping techniques for general networks.

4.3.1 *Delay Minimization.* For depth and general static delay minimization, mapping for each node can be optimized independently without worrying about logic sharing, as logic can be duplicated as needed. Therefore the depth optimal mapping of node v depends only on the mapping of nodes in N_v. Because a mapping of N_v consists of LUT_v and a mapping of $N_v - LUT_v$, an optimal mapping of N_v chooses the "best" LUT_v to minimize the delay of the optimal mapping of $N_v - LUT_v$, using dynamic programming. We present several algorithms that compute the "best" LUT_v for each node v. All of them assign a *label* for each node in topological order to guide the dynamic programming procedure and determine LUT_v for each node v.

The first method is called *dag-map*, proposed in Cong et al. [1992b] based on a classical labeling algorithm called *Lawler's labeling* [Lawler et al. 1969]. Lawler's labeling is a monotonic labeling procedure, in the sense that the labels along any path from a PI to a PO are nondecreasing with $l(v) = 0$ for any PI node v. The rule is very simple. For each nonPI node v, let p be the largest label of the nodes in $input(v)$. Then $l(v) = p$ if the set of nodes $w \in N_v$ with label $l(w) = p$ form a K-feasible cone of v; otherwise $l(v) = p + 1$. Given such a labeling, the LUT of v will be $LUT_v = \{w \mid w \in N_v, l(w) = l(v)\}$, which is K-feasible according to the labeling rule, and has depth $l(v)$. An example is shown in Figure 9(b). Note that *dag-map* requires the input network to be K-bounded in order to guarantee a mapping solution.

This simple method has a time complexity of $O(n^2)$ for a network of n nodes. But the depth of its mapping solution is optimal only if the network is *monotonic* under LUT mapping, namely, if a cone C_w is not K-feasible, then any larger cone C_v containing C_w cannot be K-feasible [Cong et al. 1992b]. However, general networks are not monotonic under LUT mapping due to the existence of reconvergent fanout paths. Consequently, *dag-map* cannot guarantee depth optimality [see Figure 9(c)].

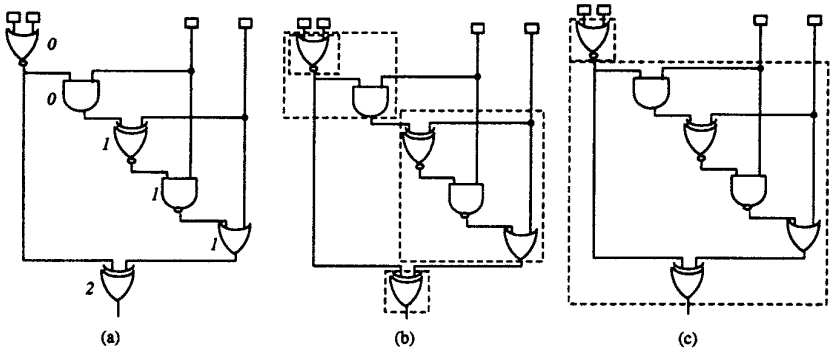

Fig. 9. General network mapping using Lawler's labeling for $K = 3$: (a) original network and labels; (b) *dag-map* mapping result of depth 2; (c) optimal mapping of depth 1.

The suboptimality of Lawler's labeling is due to the fact that when selecting LUT_v, it only looks at a neighborhood of v. This drawback was overcome by the *flowmap* method proposed in Cong and Ding [1992]. It formulates the problem of finding LUT_v as computing a *minimum height K-feasible cut* $(X, \overline{X})$ of N_v, where the height $h(X, \overline{X})$ is defined to be the largest label of nodes in X. According to the labeling rule of *flowmap*, the label of each PI is still 0, and the label of a nonPI node v is $l(v) = h(X, \overline{X}) + 1$. It can be shown that the label defined in such a way is actually the minimum depth of v in *any* mapping solution, and $LUT_v = \overline{X}$ clearly gives a solution that realizes the depth, and thus is a depth-optimal mapping solution.

The key step is then to compute a minimum height K-feasible cut for each node. It was shown in Cong and Ding [1992] that for node v and any node $w \in input(v)$, $l(v) \geq l(w)$. Moreover, if w has the maximum label among the nodes in $input(v)$, then $l(v) \leq l(w) + 1$. Therefore *flowmap* uses the following strategy. First, all nodes in N_v with labels equal to $l(w)$ are collapsed into v to obtain a reduced network N_v' [see Figure 10(b)–(c)]. This guarantees that any cut in N_v' will have height at most $l(w) - 1$. If a min-cut $(X, \overline{X})$ in N_v' is K-feasible, it will be of minimum height in N_v, and *flowmap* assigns $l(v) = l(w)$ and $LUT_v = \overline{X}$ [see Figure 10(c)–(d)]. If no K-feasible cut exists in N_v', the minimum height of a K-feasible cut in N_v must be $l(w)$. In this case, *flowmap* assigns $l(v) = l(w) + 1$ and $LUT_v = \{v\}$, given that v is K-feasible.

The min-cut in N_v' can be computed using the *node-splitting* transformation and *maximum flow computation* (Details can be found in Cong and Ding [1992]). Although originally presented only for K-bounded networks, *flowmap* is applicable to any K-mappable network. If node v is not K-feasible and no K-feasible cut exists in N_v', LUT_v can be determined by a min-cut in N_v, which is K-feasible if the network is K-mappable. The time complexity of *flowmap* is $O(Kmn)$ for a network of n nodes and m edges.

This algorithm can be extended to general static delay models. The dynamic programming procedure, as well as the minimum height K-feasible cut formulation, can still be applied. Only the cut computation

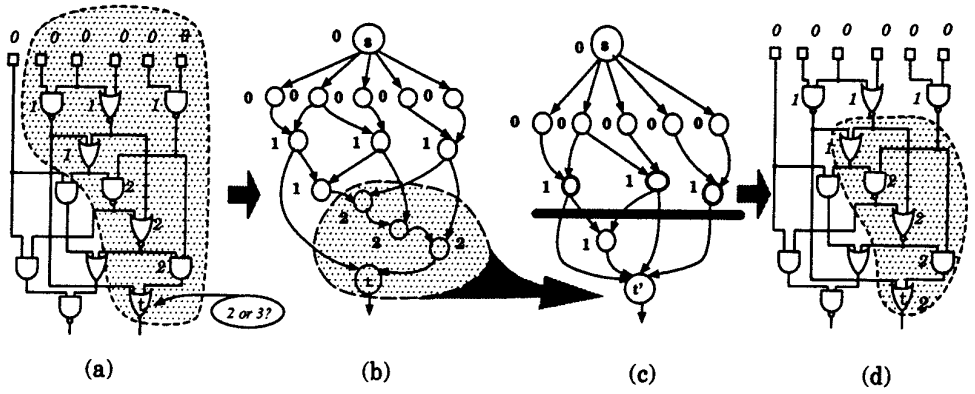

Fig. 10. Example of depth optimal mapping of general network for $K = 3$: (a) original network and labels of nodes in N_t; (b) transform into flow network; (c) collapse nodes with largest labels, compute K-feasible cut; (d) get $l(t)$ and LUT_t.

requires modifications. For net-delay models, we label each node v with the delay of the optimal mapping of N_v, including the net-delay on the output of v. Because each net can have arbitrary delay, the labeling is no longer monotonic. However, as the height of any K-feasible cut must be equal to the label of some node in N_v, one can use a binary search over the node labels to compute a minimum height K-feasible cut (see Cong et al. [1993] for details). The resulting time complexity is $(Kmn \log n)$, as the number of labels is bounded by the number of nodes. For edge-delay models, different fanout branches of a net may have different delays, resulting in multiple labels for node v, each being the sum of the delay of the optimal mapping of N_v and an edge-delay. Therefore, for each target height h, a node may have some labels larger than h but others not. Such a node can be *partially split* during the flow network construction for min-cut computation (for details see Yang and Wong [1994]). The resulting complexity is $O(Kmn \log m)$.

Under dynamic delay models, the delay of a net is linked with its structure in the mapping solution, and different branches of a multifanout node will interact. As a result, the optimal mapping of v depends not only on the optimal mapping of nodes in N_v, but also on that of nodes outside N_v. This prevents the use of the dynamic programming technique we have been using so far. In Cong and Ding [1994a], it was shown that the delay-optimal mapping problem under the dynamic nominal delay model (Section 2.3) is NP-hard. The proof transforms the well-known NP-complete problem *3SAT* [Garey and Johnson 1979] into the decision version of the dynamic nominal delay minimization problem in two ways. For general networks, the decision of the truth value assignment for a variable in 3SAT is linked to the decision of duplicating a two-fanout node. Based on this reduction, it is shown that for $K \geq 5$, the LUT mapping problem for dynamic nominal delay minimization is NP-hard. For lead-DAGs, the decision of the truth value assignment for a variable in 3SAT is linked to the decision of merging two fanouts of a node. Based on this reduction, it is shown that for $K \geq 6$, the LUT mapping problem of dynamic nominal delay

minimization for leaf-DAGs is NP-hard. This also implies that the duplication-free mapping on general networks for dynamic nominal delay minimization is NP-hard for $K \geq 6$. A heuristic was used in Cong and Ding [1994a] to incrementally adjust the static delay based on the dynamic nominal delay as the mapping process proceeds.

4.3.2 *Area Minimization.* Unlike delay optimal mapping, area optimal mapping cannot be carried out independently in each N_v, as the LUT sharing among overlapped subnetworks must be considered. Determining whether to duplicate a node of multiple fanouts (to be covered by LUTs of other nodes) or implement it as an LUT (to be shared as input to other LUTs) is a difficult problem. In fact, it was shown in Levin and Pinter [1993] that for $K = 4$ the problem of area-optimal LUT mapping is NP-hard. This result was further generalized in Farrahi and Sarrafzadeh [1994b] to $K \geq 5$. Both proofs transform the 3SAT problem to the decision version of the area-optimal mapping problem by linking the decision of the truth value assignment for a variable in 3SAT with the decision of the duplication of a node in the mapping problem. The only case in which area-optimal mapping is possible in polynomial time is when the network is *K-exact*, that is, it consists of K-input nodes only [Thakur and Wong 1995]. The difficulty of logic duplication, as indicated in this proof, is one reason that the study of duplication-free mapping for area minimization is necessary and interesting. On the other hand, it is also an indication that duplication-free mapping alone will not achieve optimality. Therefore, in practice, a second-phase mapping that exploits logic duplication often follows duplication-free mapping (e.g., in Cong and Ding [1993a]).

Given that the area-optimal mapping problem is NP-hard, it is solved either by solution space enumeration, which will have exponential time complexity, or by heuristics. There are two dimensions in the solution space: each solution has to select a subset of nodes to be implemented by LUTs, and each node has to select a K-feasible cone to be covered by its LUT implementation. We classify the mapping techniques for area minimization based on the order in which these dimensions are considered.

Node Selection Based Enumeration. Node selection based enumeration generates all node subsets for LUT implementation, and for each selected subset determines the LUT implementation of each node. This approach was first used in Woo [1991], in the form of *edge visibility*, where mapping for multi-output LUTs was considered. A subset of edges is first selected to be visible, and the invisible edges are collapsed to determine the coverage of the LUTs whose outputs are visible. For single-output LUT mapping, we can select a subset S of visible nodes, and S defines a feasible solution if and only if S contains all PO nodes, and, for each node $v \in S$, there is a K-feasible cone C_v such that each nonPI node $w \in input(C_v)$ is also in S. The existence of such a K-feasible cone for each $v \in S$ can be tested using the network flow based method as used in Cong and Ding [1994a] and Cong and Hwang [1995a]. The enumeration can be carried out using branch-and-bound to improve efficiency.

Node Covering Based Enumeration. Node covering based enumeration first generates all LUT implementations of the nodes, and then selects a subset of them to implement. The approach taken in Murgai et al. [1990] can be viewed as node covering based enumeration. For each node v, it produces all the possible LUT implementations (called *supernodes*) of v. Then it chooses a subset of the supernodes to form a mapping solution under the conditions that (1) each node must be covered in some supernode, and (2) if one supernode is chosen, each of its inputs must be a primary input or generated by another chosen supernode. The selection procedure is formulated as a *binate covering* problem,[8] which is NP-hard and is solved either exactly (by enumeration) or heuristically [De Micheli 1994].

Integer Linear Programming. Both node selection and node covering problems can be formulated in an *integer linear programming*[9] formulation as recently proposed in Chowdhary and Hayes [1995]. Each node v is associated with a variable $e(v) \in \{0, 1\}$, where $e(v) = 1$ if and only if v is visible in the mapping solution; and with a variable $s(v)$, indicating the maximum input size of an LUT containing v. Moreover, each pair of nodes u and v where some fanouts of u reconverge at v, are associated with a variable $r(u, v)$. The objective is to minimize $e(v)$, that is, the total number of LUTs, under a set of linear constraints that specify the boundary conditions (all PIs are visible and all POs need to be implemented by LUTs), LUT size constraints, and LUT size evaluation with consideration of reconvergent paths in the network. The numbers of variables and constraints may reach $O(n^2)$ for a network of n nodes, due to the existence of a possibly quadratic number of reconvergent paths. In Chowdhary and Hayes [1995] the integer linear programming problem is solved using branch-and-bound for implicitly enumerating the values of integer variables. At each step of enumeration, a subset of variables is fixed and other variables are relaxed to real valued variables. Then a *linear programming* problem is solved, and its solution is used to further the branch-and-bound search.

Node Selection Based Heuristics. For simplicity we regard the heuristic mapping methods for area minimization as approximations to the first two enumeration methods, and classify them accordingly. Note that many

---

[8] The binate covering problem can be stated as follows: Given a finite set S, a collection C of its subsets, and a relation defined on C, namely, $\rightarrow: C \times C$, we want to select the minimum number of subsets from C such that they cover S, and if a subset s is selected, then for any subset s' satisfying $s \rightarrow s'$, s' must also be selected. The covering problem can be transformed into the well-known *SAT* problem [Garey and Johnson 1979] by converting each element e of S into a clause $(s_{e1} + \cdots + s_{ek})$, where $s_{e1}, \cdots, s_{ek}$ are the subsets in C that contain e, and converting each relation $s_i \rightarrow s_j$ into a clause $(\overline{s_i} + s_j)$. The word *binate* in the name comes from the appearance of complemented variables in the second type of clauses.

[9] The integer linear programming problem is to minimize a linear expression $a_1x_1 + \cdots + a_nx_n$ under a set of constraints $c_{i1}x_1 + \cdots + c_{in}x_n \leq b_i$ for $1 \leq i \leq m$, where $x_1, \ldots, x_n$ are integer variables. This problem remains NP-hard even when all variables and coefficients are limited in $\{0, 1\}$ (called the *zero-one integer programming*) [Garey and Johnson 1979].

heuristic mapping methods are approximations of both. Also, most of these heuristics were developed independently of the preceding enumeration methods.

Many node selection based heuristics carry out a *marking* procedure in a traversal of the network in topological order, and a node is marked if it is selected for LUT implementation. For each nonPI node v, a special cone SC_v is defined as the union of $\{v\}$ and the cones SC_w of the unmarked nonPI nodes $w \in input(v)$. The input set $input(SC_v)$, consisting of only PIs and marked nodes, is called the *signal set* of v, which will be the inputs to LUT_v if SC_v is used to implement v. When v is visited, if SC_v is K-feasible, no new nodes will be marked. If SC_v is infeasible, some nodes $u \in SC_v$ will be marked, and consequently SC_u will be excluded from the union that forms SC_v. This may reduce $|input(SC_v)|$. By marking one or more nodes, SC_v will eventually become K-feasible. Once all nodes are visited, the POs and marked nodes will be implemented by LUTs. There are various ways to select the nodes to be marked. (See, for example, Farrahi and Sarrafzadeh [1994b], Groh [1991], Hwang et al. [1994], Kapoor [1994], Karplus [1991], and Levin and Pinter [1993]). A common idea is to encourage signal sharing (i.e., by marking nodes that have multiple fanouts to unvisited nodes), as this will allow fewer nodes to be marked.

Another approach to node selection is to use a *genetic* algorithm. A node subset can be represented by a bit string where each bit represents a node and a bit of value 1 represents a selected node. Standard genetic operations such as *crossover* and *mutation* can be applied on the bit strings to evolve to new strings (i.e., new solutions), and *fitness functions* can be designed to reflect the optimization objective. This method was used in Kommu and Pomeranz [1993].

Node Covering Based Heuristics. Another group of heuristics can be viewed as approximation to the node covering based enumeration. There are two ways for such approximation. First, instead of enumerating *all* possible supernodes of v, one can produce only one or a few "good" supernodes. We illustrate this approach using two examples. Second, instead of computing a true binate covering of the supernodes, one can compute a simpler covering. We also give one example of this approach.

Two types of supernode generation heuristics are intuitively good. One is to pack as many nodes into each supernode as possible; the other is to share as many input signals among the supernodes as possible. The packing based approach was used in Cong and Ding [1992] in a method called *flowpack*. Similar to *flowmap*, it computes a K-feasible cut in N_v to determine LUT_v, but unlike *flowmap*, no depth constraint is considered. Instead, the objective is to find a *maximum volume* K-feasible cut, where the volume of a cut $(X, \overline{X})$ is defined to be $|\overline{X}|$, and $LUT_v = \overline{X}$. The *flowpack* algorithm starts with a min-cut of maximum volume (which is unique according to Cong and Ding [1992]), and gradually increases the cut volume as well as the cut size until a *maximal* volume K-feasible cut is obtained. The worst case complexity of each cut computation is $O(K^3m)$ where m is

the number of edges in N_v. The sharing based approach was used in the procedure called *cutmap* [Cong and Hwang 1995a]. It improves *flowpack* by computing a *minimum cost K*-feasible cut, where the cost of a cut $(X, \overline{X})$ is defined to be the sum of the costs of the nodes in $input(\overline{X})$. To promote sharing, the algorithm assigns a low cost to the nodes that are known (or predicted) to be implemented by LUTs and assigns a high cost to other nodes. It starts with an initial set of low-cost nodes that are likely to be implemented by LUTs in a good mapping solution, such as the roots of large MFFCs, and computes the LUT for each node that has to be implemented (including PO nodes and the inputs to existing LUTs) one by one using the minimum cost *K*-feasible cut. Once a high-cost node is implemented by an LUT, its cost is lowered. Because the LUTs are determined by *K*-feasible cuts, *cutmap* can also produce a depth optimal mapping solution by adding the minimum-height constraint into the cut computation for the LUTs on the critical paths. The worst case complexity of each cut computation is $O(2Kmn^{\lfloor K/2 \rfloor + 1})$, but by using a pruning theorem the actual cost can be much lower.

The preceding methods compute at most one supernode for each node, therefore the covering part is trivial. Alternatively, we can maintain more than one supernodes per node and select the best covering heuristically. For example, graph matching can be used to approximate the binate covering formulation in Murgai et al. [1990], as used in Chen et al. [1992]. In the simplest case, if each supernode is limited to cover either one node, or a pair of nodes v and w such that v is the single output of w (called *predecessor packing* in Chen et al. [1992]), a graph can be constructed using the nodes as vertices on which each supernode of two nodes defines an edge. On such a graph, a maximum matching implies a maximum number of simultaneous pairwise merge. Efficient maximum matching computation can be applied repeatedly until no more merge is possible. For a supernode of three or more nodes, a hyperedge can be formed, and a matching implies a set of simultaneous multinode merge (a heuristic algorithm for hypergraph matching is developed in Chen et al. [1992] for this purpose).

4.4 Mapping for Routability and Low Power

Very limited work has been reported on routability and power-driven LUT technology mapping. Two approaches have been taken for routability optimization: one approach uses a heuristic cost function to guide the mapping process [Schlag et al. 1992], the other combines mapping with placement or even routing [Bhat and Hill 1992; Chen et al. 1993; Togowa et al. 1994]. For power minimization in LUT mapping, it was shown recently in Farrahi and Sarrafzadeh [1994a] that under a power dissipation model based on load capacitance and transition frequency, the problem is NP-hard. A node selection based heuristic was proposed by Farrahi and Sarrafzadeh [1994a].

5. REVIEW OF EXISTING ALGORITHMS AND SYSTEMS

In this section we review the existing algorithms and systems for LUT logic synthesis, which are built upon one or several of the techniques presented in the two preceding sections. Despite our effort, our collection may not be complete.[10] Due to space limitations, the review is very brief. But we hope it provides a fairly comprehensive and up-to-date reference source for interested readers.

Because many algorithms and systems use more than one type of optimization technique and have multiple optimization objectives, a strict classification is difficult. We choose to group the algorithms and systems of similar style or origin together in our presentation to help the readers understand the evolution of the ideas. For each algorithm and system, we present the assumption on input representation, the types of operations used and the organization of these operations, the primary and secondary optimization objectives, as well as the architecture-specific consideration and/or interaction with other design phases (if any).

5.1 The Chortle Family

One of the earliest mapping algorithms, the original Chortle [Francis et al. 1990] takes a simple-gate network as input and partitions it into leaf-DAGs. Infeasible nodes are first decomposed into feasible ones either optimally or heuristically. Then each leaf-DAG is mapped separately as a tree for area minimization, using the dynamic programming technique by enumerating all possible LUT implementations of the root node.

This algorithm later evolved into Chortle-crf [Francis et al. 1991b], which has significantly better performance and solution quality. In Chortle-crf, decomposition and technology mapping are combined in a bin-packing procedure (Section 3.1.3) using the FFD heuristic, which is much faster than Chortle, and is optimal for $K \leq 5$. Chortle-crf also exploits the reconvergence of the leaf-DAG inputs using the MSD heuristic (see Section 3.1.4), as well as the replication of the root LUT of a leaf-DAG when it can be merged into its fanout LUTs. As a result, it reduces the area by 14% when compared with the original Chortle algorithm.

The idea in Chortle-crf was then extended to depth minimization in the Chortle-d algorithm [Francis et al. 1991a] (Section 3.1.4). In addition, it minimizes area as a secondary objective by using area-optimal node decomposition along noncritical paths and depth-optimal node decomposition along critical paths, as well as predecessor packing (Section 4.3.2). The mapping solutions of Chortle-d use an average of 35% fewer levels of LUTs than those of Chortle-crf, at the cost of an average of 59% larger area.

Both Chortle-crf and Chortle-d have very efficient implementations. The Chortle algorithms have solved the optimal mapping problem for an *un-*

[10] Noticeably, the commercial FPGA synthesis systems are absent, because their detailed algorithms are generally unknown to the public.

bounded tree, but *a prior* tree partitioning often compromises the mapping quality.

5.2 The MIS-pga Family

Another early algorithm, the original MIS-pga [Murgai et al. 1990], was an extension of the UC Berkeley MIS-II logic synthesis system [Brayton et al. 1987] to FPGA synthesis. It is applicable to general networks for area minimization. In the logic optimization step, it first uses Roth-Karp decomposition, kernel extraction, and AND-OR decomposition to decompose the network into a K-bounded one. Then it collapses nodes into their fanouts, while maintaining K-feasibility, in a heuristically determined order. In the technology mapping step it uses node covering based enumeration (see Section 4.4.2). It also includes a postprocessing step using maximum matching to merge pairs of LUTs into Xilinx XC3000 (or, equivalently, AT&T ATT3000) cells (called CLBs) which can either implement one 5-LUT, or two 4-LUTs with a total of five distinct inputs [Xilinx 1994]. Such a procedure has since been used in many other algorithms and systems.

A subsequent improvement was referred to as MIS-pga(new) [Murgai et al. 1991b], where the logic optimization procedure was substantially enhanced with more decomposition techniques including cube-packing, cofactoring, and cube partitioning. All decomposition methods are tried and the best result is kept. The K-feasibility constraint during node collapsing is relaxed, and the collapsed nodes that are not K-feasible are redecomposed according to a cube-packing based quick cost estimation. This way the entire network may be collapsed if the number of PIs is small. The binate covering in the mapping step is reformulated to include two-output supernodes to facilitate better XC3000 CLB mapping. These enhancements result in an area reduction of 28.2% compared with MIS-pga.

Another member in this family is MIS-pga(delay) [Murgai et al. 1991a] for delay minimization. In topological order, the algorithm tries to collapse each critical node into its critical fanouts. If such collapse is K-feasible, or can be made K-feasible by decomposition without increasing level, it is performed. This is repeated until no more collapse is possible. When the number of PIs is small, it also tries other approaches, that is, to collapse the entire network into a two-level one and use cofactoring and Roth-Karp decomposition, respectively, to get two new K-bounded networks. The best result of all applicable approaches is chosen for technology mapping using a heuristic binate covering. The mapping solution is further improved by a pseudoplacement phase, in which the LUTs are iteratively placed in a 2-D grid using *simulated annealing* [Kirkpatrick et al. 1983]. At each iteraction, MIS-pga(delay) identifies critical sections, decomposes critical nodes, updates the placement (with reduced routing congestion due to the decomposition), and performs local collapsing to reduce the number of nodes.

The MIS-pga FPGA synthesis system also provides other functionalities in addition to those in the preceding algorithms, such as sequential

synthesis for LUT based FPGAs [Murgai et al. 1993b] and synthesis for nonLUT based FPGAs [Murgai et al. 1992]. As part of the MIS-II system, it has great flexibility in combining various logic synthesis operations in its course of optimization, although the time-consuming nature of these operations often limits the space of exploration.

5.3 The TechMap Family

The TechMap algorithms represent the combination and enhancement of the ideas of both the Chortle algorithms and the MIS-pga algorithms. The original TechMap algorithm for area minimization [Sawkar and Thomas 1992] works on a simple-gate network and performs combined decomposition and mapping on a general network directly, using a greedy heuristic to determine the decomposition (Section 3.1.3). It also collapses small networks and then redecomposes them by cofactoring to explore alternative network structures.

The delay minimization version TechMap-L [Sawkar and Thomas 1992], is similar to the TechMap algorithm, except that in the decomposition heuristic, priority of grouping a pair of inputs is determined based on the depth of the resulting node. When small networks are collapsed and redecomposed, cofactoring is also guided by depth. Subsequently, this algorithm was improved in TechMap-D [Sawkar and Thomas 1993] with two major enhancements: combined area and depth minimization by applying the TechMap algorithm on noncritical nodes, and the TechMap-L algorithm on critical nodes, and a better cost function that represents the tradeoff of depth, area, and input size. It also has a placement phase, but is performed separately after mapping without resynthesis.

5.4 The FlowMap Family

A notable progress in LUT logic synthesis was the development of delay-optimal technology mapping algorithms for general networks. A number of algorithms can be included in this family, all related to the FlowMap algorithm [Cong and Ding 1992, 1994c], which was the first polynomial-time depth-optimal mapping algorithm for general K-mappable networks.

The predecessor of FlowMap was the DAG-Map algorithm [Cong et al. 1992b]. It first transforms a general network into a depth-minimum two-bounded simple-gate network using AND-OR and Huffman tree decompositions. Then it maps the network using the *dag-map* algorithm (Section 4.3.1). Finally, it improves the mapping solution using two postprocessing operations, namely, *gate decomposition* (Section 3.5.1) and *predecessor packing* (Section 4.3.2), to minimize the number of LUTs based on a maximum matching formulation.

The original FlowMap algorithm [Cong and Ding 1992, 1994c] goes through the same logic optimization steps as DAG-Map, but uses the *flowmap* procedure (Section 4.3.1) for mapping, and includes one more postprocessing step using the *flowpack* operation (Section 4.3.2). The minimum height K-feasible cut computation used in *flowmap* guarantees

depth-optimal mapping for general K-bounded networks. The FlowMap mapping results were shown to be superior to those of Chortle-d, DAG-Map, and MIS-pga(delay)—these algorithms use 9–50% more LUTs with up to 7% larger depth on average compared to FlowMap [Cong and Ding 1992].

The FlowMap algorithm has inspired a number of follow-up algorithms with various enhancements. The CutMap algorithm [Cong and Hwang 1995a] considers area minimization during depth-optimal mapping. It replaces the minimum height K-feasible cuts computed by FlowMap with minimum-cost K-feasible cuts of bounded heights, where the cost of a cut measures the potential area increase (Section 4.3.2). A 13% area reduction was reported in Cong and Hwang [1995a] compared with FlowMap. In a different approach, the FlowMap-r algorithm [Cong and Ding 1993a, 1994b] enhances FlowMap with more powerful postprocessing operations. After the mapping solution is obtained using the *flowmap* procedure, FlowMap-r identifies the critical paths according to a given depth constraint, and then partially or fully undoes the depth-optimal mapping along the noncritical paths without violating the depth constraints using a set of *depth relaxation* heuristics. The resulting network is then remapped using the duplication-free *df-map* algorithm (Section 4.2) followed by the FlowMap postprocessing steps to further exploit beneficial logic duplication. The FlowMap-r algorithm saves an average of 10% LUTs compared to FlowMap without compromising depth optimality [Cong and Ding 1993a]. Another interesting feature of FlowMap-r is that it can produce a *spectrum* of mapping solutions for a given design by gradually relaxing the depth constraint and producing an area-minimized solution for each depth bound. This provides area-depth tradeoff in the selection of mapping solutions. Also using the relaxation concept, the Sweep algorithm [Shin and Kim 1995] first uses Lawler's labeling (i.e., *dag-map*) to determine the node labels and perform the initial mapping. Then it goes through a number of *sweeping* iterations to change the node labels (and thus the mapping solution) in order to reduce the *cost* of the mapping solution, which is measured by a heuristic function of the total number of LUTs and the total number of LUT inputs. Finally a greedy packing procedure is used as postprocessing.

Improving FlowMap from a different angle, the FlowSYN algorithm [Cong and Ding 1993b] aims at further enhancement of the depth minimization by incorporating logic optimization into the technology mapping procedure. When computing the node label of v, if the largest node label in N_v is p and N_v does not have a K-feasible cut of height $p - 1$, *flowmap* would assign $l(v) = p + 1$ (Section 4.4.1). However, FlowSYN finds a cut $C_i = (X_i, \overline{X}_i)$ of height $p - i$ (where $2 \leq i \leq p - 1$), and tries to redecompose $\overline{X}_i$ in such a way that after the decomposition, the distance from any node $w \in input(\overline{X}_i)$ to v will be no more than $p - l(w)$. Such a decomposition, if it exists, will give a mapping solution of depth p for N_v and we can still have $l(v) = p$. FlowSYN tries the redecomposition in the order $i = 2, 3, \ldots, p - 1$, and uses the OBDD based functional decomposition with the preference of choosing the bound set from the nodes with smaller depths. For area

minimization, a redecomposition that will reduce the area but not the depth is also accepted as long as it does not increase the critical path depth. A saving of 20% in area and 13% in depth compared with FlowMap was reported in Cong and Ding [1993b].

Another direction of improving FlowMap is to extend the delay model from unit delay to general delays. First, general static net-delay models were used in the FlowMap-d algorithm [Cong et al. 1993, 1994]. It uses binary search to determine the minimum height K-feasible cut. The delay assignment can be determined by an iterative mapping and placement procedure [Gao et al. 1993] that estimates the delay of each net after placement. A pseudodynamic delay assignment procedure was proposed in Cong and Ding [1995] based on the nominal delay model. The FlowMap algorithm has also been generalized to static edge-delay models. The Bias-Clus algorithm in Mathur and Liu [1994] computes K-feasible cuts according to a *biased* topological order that puts nodes on longer paths closer to v, so that they are more likely covered by LUT_v. But Bias-Clus does not guarantee the optimality of its mapping under the edge-delay model. Later on, this problem was optimally solved by the Edge-Map algorithm [Yang and Wong 1994], which modifies the node-splitting operation in the flow network construction of FlowMap-d so that each node can carry multiple labels during the cut computation.

The Huffman-tree based decomposition was used by FlowMap and its variations for decomposing input networks when they are not K-mappable. It minimizes the number of levels in the decomposed network, but not the depth of the LUT *mapping solution*. An improved algorithm, named DOGMA, was developed to compute better structural gate decomposition of general networks for LUT mapping (Section 3.1.4) [Cong and Hwang 1996]. Because DOGMA considers the depth-optimal node labeling process for K-mappable networks used in FlowMap, it usually leads to better mapping results in terms of both depth and LUT count.

The FlowMap algorithm and its successors, including FlowMap-r, CutMap, and FlowSYN, have been incorporated into a general logic synthesis system for LUT based FPGAs, named RASP [Cong et al. 1996]. RASP consists of a core with a set of LUT synthesis algorithms, together with a set of architecture-specific technology mapping routines to map a generic LUT network to programmable logic blocks in various LUT based FPGA architectures, so that it can produce designs optimized for various LUT based FPGA architectures, and can quickly adapt to new LUT based FPGA architectures.

5.5 Partitioning-Enumeration Based Mappers

Although many systems and algorithms use enumeration at some steps to compute intermediate solutions during logic optimization or mapping, two mappers rely on direct application of branch-and-bound enumeration to find optimal mapping solutions, often after partitioning a large network into smaller portions. An early algorithm named Vismap [Woo 1991] uses

the *edge visibility* concept to characterize a solution by the edges not covered inside the logic cells of the target XC3000/ATT3000 CLBs. Once a set of visible edges is chosen, invisible edges are collapsed to reduce the network into a mapped one. By enumerating the solutions, it is capable of finding the area-optimal mapping; but because the number of candidates is very large even with branch-and-bound pruning, Vismap first partitions the network into smaller regions, and then finds optimal mapping for each region.

A more recent MILP approach by Chowdhary and Hayes [1995] uses the *mixed integer linear programming* formulation, which can be applied to area minimization as we introduced in Section 4.3.2, as well as other objectives such as depth minimization with area constraint. The MILP formulation was solved using branch-and-bound enumeration. Due to its high complexity, large networks are first partitioned based on their high-level structures before each portion is mapped, which was reported to yield significant speedup with only marginal loss of quality.

5.6 Node Selection Based Heuristic Mappers

In this section we present several heuristic mapping algorithms for area minimization, which are mostly based on the node selection heuristic in Section 4.3.2. The common core of most of these algorithms is the marking procedure that marks visible nodes in topological order, to achieve the feasibility of the input signal set $input(SC_v)$ for each node v (Section 4.3.2). These algorithms differ in terms of the criteria used for choosing nodes to mark.

The Level-Map algorithm [Farrahi and Sarrafzadeh 1994b] works on a K-bounded network, and when node v is processed and $|input(SC_v)| > K$, the unmarked nodes w in $input(v)$ are marked in decreasing order of the cost function $p(w) = |input(\mathrm{SC}_w)| + \delta|output(w)|$, where δ is called the *fanout factor*, until SC_v becomes K-feasible. This is a generalization of *tree-map* (Section 4.1) where $\delta = 0$. By having $\delta > 0$, nodes with multiple fanouts are given higher priority to be marked. (A variation of Level-Map, called Level-Map-p, was also proposed in Farrahi and Sarrafzadeh [1994a] for power minimization, where the marking order is based on two cost functions.)

The same cost function has also been used in the Factor-Map [Hwang et al. 1994]. It first recursively performs a restricted form of functional decomposition $f(X) = \bigoplus_{1 \le i \le m} f_{l_i}(X_l) f_{r_i}(X_r)$, where $X_l \cup X_r = X$ and $X_l \cap X_r = \emptyset$, to make the network a tree of K-feasible XOR and AND gates. Then a similar marking procedure is applied. The TeXmap algorithm in Kapoor [1994] marks $w \in \mathrm{input}(v)$ for v, in the increasing order of $|I_w| - |D_w|$, where I_w is the set of nodes $u \in output(w)$ such that $|input(SC_u)|$ increases after w is marked,[11] and D_w is the set of nodes $u \in$

[11] This will happen only when $input(SC_u)$ becomes redundant, and can be easily avoided.

$output(w)$ such that $|input(\mathrm{SC}_u)|$ decreases after w is marked (except that a fanout-free input is marked last.)

Other algorithms also consider the nodes outside $input(v)$ when marking for node v. The algorithm in Groh [1991] marks a node $w \in SC_v$ with minimum cost $\Sigma_{u \epsilon I_w} \log_K (|input(SC_u)|)$, where $I_w = \{ u \mid input(SC_u) > K$ after w is marked$\}$, after converting the input network into a NAND2 network. The Xmap algorithm [Karplus 1991] first converts the network into an ITE, and then marks the ITE in a multiphase approach: it first marks the nodes $w \in input(v)$ with $|input(SC_w)| > h$, where h is a threshold value. If this is not enough, recursively the nodes $u \in input(w)$ of the nodes $w \in input(v)$ with $|input(SC_w)| > h + 1$ are marked, and so on. Finally, if these are still not enough, the unmarked nodes $w \in input(v)$ are marked in decreasing order of $|input(SC_w)|$.

The three algorithms proposed in Levin and Pinter [1993] use marking procedures for 4-LUT mapping that do not follow the topological order. Each of the algorithms first decomposes each infeasible node using minimum tree decomposition on simple gates and cofactoring on complex gates, then implements each four-input node with a 4-LUT and marks the node and its inputs. Then unimplemented POs and marked nodes v have their SC_v computed, and each of these algorithms marks nodes in $L = \cup_{|input(SC_v)|>4} (SC_v - \{v\})$ using a different heuristic. Once a new node is marked, L is updated. The procedure ends when $L = \varnothing$.

Finally, the GAFPGA algorithm [Kommu and Pomeranz 1993] uses a genetic algorithm to reduce the number of nodes to implement. It performs both simple-gate decomposition and technology mapping: each mapping solution by two bit-strings, one for the fully decomposed two-input network in which selected nodes are marked (by the bits of value 1), and the other for the record of the decomposition. Crossover and mutation operations are implemented and invalid offspring are corrected by randomly marking new nodes.

5.7 Decision Diagram Based Algorithms

The Xmap algorithm [Karplus 1991] presented in the preceding subsection is the first to use decision diagram representation for LUT mapping. It converts the input network in an ITE and maps the ITE directly. The algorithm in Besson et al. [1994] also maps ITE directly. Each node is first represented by its OBDD (if the OBDD is too large, the node is decomposed first), which is minimized by variable ordering heuristics based on an improved lexicalgraphical approach. The OBDDs are then connected to form an ITE network and mapped into LUTs. In the algorithm proposed in Schubert et al. [1994], the *Ordered Functional Decision Diagram* (OFDD), based on Davio expansion, is used to represent the node functions, and is minimized by variable ordering heuristics and inverter insertion. The connected OFDD network is mapped for depth reduction by visiting each node in reversed topological order, collapsing its input(s) with largest level(s) into the node without violating K-feasibility.

Many algorithms use decision diagrams, in particular OBDDs, for logic optimization. In Sasao [1993], an algorithm was proposed to implement a function of seven or more variables with a network of 5-LUTs. It decomposes an m-input function using up to three $(m - 2)$-input encoding functions, so that the base function is five-feasible. If such a decomposition is impossible after trying all variable partitions, cofactoring with respect to two variables is used. In either case the $(m - 2)$-input functions are recursively decomposed if necessary. The BDD-Syn algorithm in Chang and Marek-Sadowska [1992] works on a nine-bounded network, using both OBDD based functional decomposition and OBDD based extraction and rule based reduction (Section 3.5.2) for logic optimization aiming for 5-LUTs. The resulting five-feasible network is accepted as an LUT network without an explicit mapping phase. (Another functional decomposition based algorithm that does not have an explicit mapping phase is the TRADE algorithm [Wan and Perkowski 1992], which uses extended Ashenhurst decomposition for incompletely specified, multi-output functions by superimposing the partition matrices, properly assigning don't-cares, and partition matrix patching, as in Section 3.3.2.)

OBDD based extraction was used in the Catamount algorithm [Stanion and Sechen 1995] for multilevel decomposition into LUT networks by enumerating different extraction schemes and accepting the *best* extractions at each level according to a cost function that resembles a lower bound of the final decomposition size if this extraction scheme is accepted (Section 3.2.3).

The algorithm in Lai et al. [1993a] uses OBDD based functional decomposition for direct mapping to Xilinx XC4000 CLBs. Each XC4000 CLB implements a function of the form $f(x_1, g(y_1, \ldots, y_4), h(z_1, \ldots, z_r))$ with at most two of f, g, and h available as the outputs [Xilinx 1994]. The algorithm tests variable partitions against the XC4000 configuration based on a set of rules by swapping bound set variables in the OBDD. This is generalized in the FGMap algorithm [Lai et al. 1993b] to *two-layer* decomposition, where for a function that is decomposable in the form $f(g(Y), h(Z), \ldots)$ under two bound sets Y and Z, the algorithm looks for a *mergeable* pair of encoding functions $g(Y)$ and $h(Z)$ such that they can be packed to form a single output function $p(g(Y), h(Z))$. If so, the two functions can be implemented by an XC4000 CLB. To improve the chance of such a decomposition, FGMap exploits different encodings of g and h for each partition Y and Z, based on a set of existence conditions and a greedy algorithm.

OBDD based multi-output functional decomposition was implemented in the FGSyn algorithm [Lai et al. 1994], where the outputs are heuristically partitioned into groups such that each group can share strict-coding type encoding functions. The IMODEC algorithm [Wurth et al. 1995] also considers shared nonstrict-coding. It uses a more powerful decomposition method by enumerating *preferable functions* to find shared encoding functions (Section 3.3.2).

A recent system by Sawada et al. [1995], also based on OBDD representation, uses decomposition and also simplification (in the form of *resubstitution*, which looks for a permissible function of a node using a given set of nodes as inputs). For each node function f_i that is not K-bounded, it uses functional decomposition to find a set of K-feasible encoding functions $\tilde{\alpha}_i$. Then it tries the resubstitution of each $\tilde{\alpha}_i$ into each f_j, which (if successful) results in a new function g_{ij}. The *gain* of $\tilde{\alpha}_i$, defined by $(|\ input(g_{ij})\ | - |\ input(f_j)|)$ over all its successful resubstitutions, is computed and the best $\tilde{\alpha}_b$ is chosen. As a result, f_b is decomposed with $\tilde{\alpha}_b$, and each f_j that can be resubstituted with $\tilde{\alpha}_b$ is replaced by g_{bj}. The procedure repeats until all nodes are K-bounded, and each K-bounded node is given an LUT. Optionally, resubstitution of each PO into each other PO can also be tried before the foregoing procedure.

5.8 Mapping Enhancement Algorithms

The objective of these algorithms is not to produce a complete mapping solution, but to enhance existing algorithms, either by producing a better initial network, or by further optimization of a mapped LUT network. Two algorithms were proposed to improve MIS-pga by more effective logic optimization. The xl-map algorithm in Fujita and Matsunaga [1991] introduces a preprocessing step that uses the minimal dependence set based simplification (Section 3.5.3) to optimize the input network to MIS-pga. The AFLO algorithm in Lu et al. [1994] modifies the cost criteria used in MIS-II to trade off support minimization and literal minimization in simplification, extraction, and substitution operations to prepare input for MIS-pga. Two other algorithms focus on the reduction of a mapped LUT network. The RENO-FPGA algorithm [Chen 1992] considers fanout-free LUT elimination and LUT fanin reduction (for further merging) using minimal dependence set base simplification (Section 3.5.3). The MR algorithm in Chen and Cong [1992] reduces fanout-free LUTs more systematically by computing a *maximum acyclic independent set* of fanout-free LUTs that can be simultaneously removed or replaced.

5.9 Library Based Mapping Algorithms

Although the library based mapping approach is generally regarded as not suitable for LUT based FPGAs due to the potential large size of the library, efforts have also been made to reduce the library size, by merging equivalent patterns (based on the fact that input signals of an LUT are completely symmetric) and/or limiting to the most frequently used K-variable functions, and the like. Libraries of manageable size for practical values of K were constructed in Bhat [1993] and Trevillyan [1993]. One reason for such an approach (which cannot scale up with K) is to use existing library based technology mapping tools which are often part of a larger system and are too costly to replace [Trevillyan 1993].

5.10 Architecture Specific Algorithms

Although most algorithms and systems generate LUT networks as their results, many also have a postprocessing phase to convert the LUT networks as their results, many also have a postprocessing phase to convert the LUT network into logic cells of a particular type of FPGAs. There are also algorithms that directly generate such architecture-specific FPGA logic cells (such as the Vismap algorithm [Woo 1991] in Section 5.5 and the FGMap algorithms [Lai et al. 1993a,b] in Section 5.7).

Many early mapping algorithms aimed at the XC3000/ATT3000 architecture: Vismap is an example. The Hydra algorithm [Filo et al. 1991] constructs a *shared-input graph* over the nodes in the network where each edge carries a weight equal to the number of shared inputs of the two nodes. It is used to guide simple disjunctive decompositions to produce shared nodes. AND-OR decomposition is also used to make the network feasible. Finally, the network is reduced by local collapsing and covered by CLBs. The ALOE-CLB [Dresig et al. 1991] uses functional decomposition to make the network K-bounded, then greedily covers it with XC3000 CLBs. An algorithm of similar flavor, but aimed at the XC4000 CLB, was reported in Weinmann and Rosenstiel [1994], which uses the decomposition methods of MIS-pga, together with iterative cube partitioning based on the Kernighan-Lin method, and then places the nodes into the CLBs by pattern matching.

The first routability driven algorithm, the Rmap algorithm [Schlag et al. 1992, 1994], also aims at XC3000 CLB. It first decomposes an AND-OR network into a t-bounded one, $t \leq K$, using repeated extraction for interconnection reduction followed by a balanced tree decomposition. Then it generates all possible LUT coverings of each node, and finds all potential pairings for these coverings under the XC3000 CLB constraints. Finally, it selects the pairs using a heuristic cost function reflecting the *pin-to-cell ratio*, a routability estimation.

Several mapping algorithms were developed to facilitate FPGA architecture research. They consider special features of FPGA cells. The TEMPT algorithm in Chung and Rose [1992] minimizes depth assuming the existence of *hardwires* [Chung et al. 1991]. In He and Rose [1994], a *heterogeneous LUT-mapping* algorithm was proposed for cell groups containing several LUTs of two sizes. Both algorithms are based on Chortle-crf.

5.11 Layout Based Algorithms

As the impact of placement on area, delay, and routability of the network is significant, a number of algorithms consider layout issues in logic synthesis. For example, MIS-pga(delay) [Murgai et al. 1991a], performs iterative logic optimization and placement. The Mmap algorithm [Chen et al. 1993] more tightly couples technology mapping and placement for delay minimization by direct mapping of a two-bounded network, generated by Roth-Karp decomposition, into a 2-D grid of LUTs, using a maximum weighted matching formulation to assign LUTs to their preferred locations. The

Plack algorithm [Bhat and Hill 1992] combines placement with technology mapping for routability optimization. It starts with a mapped network, places it in a 2-D grid, and then uses simulated annealing to swap logic among the LUTs based on the impact on a global routing. An integrated mapping, placement, and routing approach is also used by the Maple algorithm [Togawa et al. 1994], where repeated bipartitioning is performed to map the network into a 2-D grid of cells while maintaining routability. At each iteration the LUTs along the region boundary have been fixed, and after a partition is computed, new nodes are chosen (according to their connection with the boundary LUTs) to be implemented by LUTs and placed along the cut-line that divides the region into two for the next iteration.

The two LUT logic synthesis algorithms using the *patching* approach can also be viewed as layout based algorithms. The algorithm proposed in Fujita and Kukimoto [1992] and Kukimoto and Fujita [1992] modifies a mapped and routed LUT network to accommodate logic changes of the design by keeping the routing unchanged, and only changing the functions of the LUTs. It uses permissible function based network simplification (Section 3.5.3). On the other hand, the algorithm in Chang et al. [1994] patches the mapped network to improve routability by replacing connections in congested areas with new connections through less congested regions based on redundancy addition and removal (Section 3.5.2).

6. CONCLUSION

The increasing popularity of the technology and the unique feature of the architecture have led to intensive studies on design automation techniques for LUT based FPGAs. This article summarized various techniques for combinational logic synthesis of LUT based FPGAs, including logic optimization techniques, the technology mapping techniques, and their applications in FPGA synthesis algorithms for area, delay, routability, and power optimization. These techniques vary considerably in terms of quality and efficiency, and different ones may be suitable for different types of designs and/or optimization objectives. We hope our systematic classification and review of these techniques will help the reader to choose the best combination of these techniques for a given application, and to develop new techniques to overcome the limitations in the existing approaches.

Combinational logic synthesis is a very important step in design automation of the FPGA technology. Its potential is far from being fully exploited in current commercial CAD tools—there has been a big gap between the vendor-estimated logic density on an FPGA chip and the *usable* density achieved by the CAD tools. Although the estimated density may not be achievable for all designs, it is certainly true that there is still plenty of room for logic synthesis tools to improve. We believe that high-quality, automatic logic synthesis tools will play a more and more important role in FPGA design systems, especially as the FPGA chip capacity increases, and when more and more users design the systems at a more abstract level

using high-level description languages (HDLs). Much work remains to be done, such as the development of simple but accurate measurements for FPGA specific logic optimization, better cost models for technology mapping, efficient mapping algorithms for constrained optimization (such as delay minimization with given area and routing constraints), mapping algorithms for easy adaptation to design and/or architecture specific features and constraints, and synthesis for multiple FPGA systems. Better integration with other steps in the design process, such as high-level synthesis, sequential logic synthesis, and layout synthesis, is also key to success.

REFERENCES

AT&T MICROELECTRONICS 1995. *AT&T Field-Programmable Gate Arrays Data Book.* AT&T Corp., Berkeley Heights, NJ.

AKERS, S. B. 1978. Binary decision diagrams, *IEEE Trans. Comput. 27,* 6, 509–516.

ALLEN, D. 1992. Automatic one-hot re-encoding for FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Vienna, Austria, Aug.), 71–77.

ALTERA 1994. *Programmable Logic Devices Data Book,* Altera Corp., San Jose, CA.

ASHENHURST, R. L. 1957. The decomposition of switching functions. In *Proceedings of International Symposium on Theory of Switching.* (Harvard University, MA, Apr.), 74–116.

BECKER, B. AND DRECHSLER, R. 1995. How many decomposition types do we need. In *Proceedings of the European Design and Test Conference,* (Paris, March).

BESSON, T., BOUZOUZOU, H., LE, V. V., TIXIER, S., AND SAUCIER, G. 1994. Use of binary decision diagram for FPGA mapping. *Proceedings of ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

BHAT, N. 1993. Library-based mapping for LUT FPGAs revisited. In *Proceedings of the International Workshop on Logic Synthesis* (Tahoe City, CA, May), P9b.1–6.

BHAT, N. AND HILL, D. D. 1992. Routable technology mapping for LUT FPGAs. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 95–98.

BRACE, K., RUDELL, R. L., AND BRYANT, R. E. 1990. Efficient implementation of a BDD package. In *Proceedings of the ACM/IEEE Design Automation Conference* (Orlando, FL, June), 40–45.

BRAYTON, R. K., HACHTEL, G. D., MCMULLEN, C. T., AND SANGIOVANNI-VINCENTELLI, A. L. 1984. *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer, Hingham, MA.

BRAYTON, R. K., HACHTEL, G., AND SANGIOVANNI-VINCENTELLI, A. L. 1990. Multilevel logic synthesis. *Proc. IEEE 78,* 2, 264–300.

BRAYTON, R. K., RUDELL, R., SANGIOVANNI-VINCENTELLI, A. L., AND WANG, A. R. 1987. MIS: A multiple-level logic optimization system. *IEEE Trans. Comput. Aided Des. 6,* 6, 1062–1081.

BROWN, S. D., FRANCIS, R. J., ROSE, J., AND VRANESIC, Z. G. 1994. *Field-Programmable Gate Arrays.* Kluwer, Norwell, MA.

BRYANT, R. E. 1995. Binary decision diagrams and beyond: Enabling techniques for formal verification. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 236–243.

BRYANT, R. E. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24,* 3, 293–318.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35,* 6, 677–691.

BUTLER, K. M., ROSS, D. E., KAPUR, R., AND MERCER, M. R. 1991. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 417–420.

CHAN, P. K. AND MOURAD, S. 1994. *Digital Design Using Field Programmable Gate Arrays.* PTR Prentice-Hall, Englewood Cliffs, NJ.

CHAN, P. K., SCHLAG, M. D. F., AND ZIEN, J. Y. 1993. On routability prediction for field-programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 326–330.

CHANG, S.-C. AND MAREK-SADOWSKA, M. 1992. Technology mapping via transformations of function graphs. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 159–162.

CHANG, S.-C., CHENG, K.-T., WOO, N.-S., AND MAREK-SADOWSKA, M. 1994. Layout driven logic synthesis for FPGAs. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Diego, CA, June), 308–313.

CHEN, C.-S., TSAY, Y.-W., HWANG, T.-T., WU, A. C. H., AND LIN, Y.-L. 1993. Combining technology mapping and placement for delay-optimization in FPGA designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 123–127.

CHEN, K.-C. 1992. Logic minimization of lookup-table based FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.) 71–76.

CHEN, K.-C. AND CONG, J. 1992. Maximal reduction of lookup-table based FPGAs. In *Proceedings of the European Design Automation Conference* (Hamburg, Germany, Sept.), 224–229.

CHEN, K.-C., CONG, J., DING, Y., KAHNG, A. B., AND TRAJMAR, P. 1992. DAG-map: Graph-based FPGA technology mapping for delay optimization. *IEEE Des. Test Comput.* (Sept.), 7–20.

CHEN, K.-C., MATSUNAGA, Y., FUJITA, M., AND MUROGA, S. 1991. A resynthesis approach for network optimization. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 458–463.

CHOWDHARY, A. AND HAYES, J. P. 1995. Technology mapping for field-programmable gate arrays using integer programming. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 346–352.

CHUNG, K. AND ROSE, J. 1992. TEMPT: Technology mapping for exploration of FPGA architectures with hard-wired connections. In *Proceedings of the ACM/IEEE Design Automation Conference* (Anaheim, CA, June) 361–367.

CHUNG, K., SINGH, S., ROSE, J., AND CHOW, P. 1991. Using hierarchical logic blocks to improve the speed of FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Oxford, England, Sept.) 103–113.

CONG, J. AND DING, Y. 1995. On nominal delay minimization in LUT-based FPGA technology mapping. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays* (Monterey, CA, Feb.), 82–88.

CONG, J. AND DING, Y. 1994a. On nominal delay minimization in LUT-based FPGA technology mapping. *Integration—VLSI J. 18,* 73–94.

CONG, J. AND DING, Y. 1994b. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Trans. VLSI Syst. 2,* 2, 137–148.

CONG, J. AND DING, Y. 1994c. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. Comput. Aided Des. 13,* 1, 1–12.

CONG, J. AND DING, Y. 1993a. On area/depth trade-off in LUT-based FPGA technology mapping. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 213–218.

CONG, J. AND DING, Y. 1993b. Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 110–114.

CONG, J. AND DING, Y. 1992. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 48–53.

CONG, J., DING, Y., GAO, T., AND CHEN, K.-C. 1994. LUT-based FPGA technology mapping under arbitrary net-delay models. *Comput. Graph. 18,* 4, 137–148.

CONG, J., DING, Y., GAO, T., AND CHEN, K.-C. 1993. An optimal performance-driven technology mapping algorithm for LUT based FPGAs under arbitrary net-delay models. In *Proceedings of the International Conference on CAD and Computer Graphics* (Beijing, China, Aug.), 599–603.

CONG, J., DING, Y., KAHNG, A. B., TRAJMAR, P., AND CHEN, K.-C. 1992a. An improved graph-based FPGA technology mapping for delay optimization. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 154–158.

CONG, J. AND HWANG, Y.-Y. 1996. Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA designs. In *Proceedings of the ACM/IEEE Design Automation Conference* (Las Vegas, NV, June), 726–729.

CONG, J. AND HWANG, Y.-Y. 1995a. Simultaneous depth and area minimization in LUT-based FPGA mapping. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays* (Monterey, CA, Feb.), 68–74.

CONG, J. AND HWANG, Y.-Y. 1995b. A theory on partially dependent functional decomposition with application in LUT-based FPGA. UCLA Computer Science Department Tech. Rep. CSD-950050, Dec.

CONG, J., KAHNG, A. B., TRAJMAR, P., AND CHEN, K.-C. 1992b. Graph based FPGA technology mapping for delay optimization. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.) 77–82.

CONG, J., PECK, J., AND DING, Y. 1996. RASP: A general logic synthesis system for SRAM-based FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays* (Monterey, CA, Feb.), 137–143.

CURTIS, H. A. 1963. Generalized tree circuit—the basic building block of an extended decomposition theory. *J. ACM 10,* 3, 562–581.

CURTIS, H. A. 1961. A generalized tree circuit. *J. ACM 8,* 4, 484–496.

DETJENS, E., GANNOT, G., RUDELL, R., SANGIOVANNI-VINCENTELLI, A., AND WANG, A. R. 1987. Technology mapping in MIS. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Nov.), 116–119.

DEVADAS, S., GHOSH, A., AND KEUTZER, K. 1994. *Logic Synthesis.* McGraw-Hill, New York.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, New York.

DRESIG, F., RETTIG, O., AND BAITINGER, U. G. 1991. Logic synthesis for universal logic cells. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Oxford, England, Sept.), 181–190.

FARRAHI, A. AND SARRAFZADEH, M. 1994a. FPGA technology mapping for power minimization. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Prague, Czech Republic, Aug.), 66–77.

FARRAHI, A. AND SARRAFZADEH, M. 1994b. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. Comput. Aided Des. 13,* 11, 1319–1332.

FILO, D., YANG, J., MAILHOT, F., AND DE MICHELI, G. 1991. Technology mapping for a two-output RAM-based field programmable gate arrays. In *Proceedings of the European Conference on Design Automation* (Amsterdam, the Netherlands, Feb.), 534–538.

FRANCIS, R. J. 1992. A tutorial on logic synthesis for lookup-table based FPGAs, In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 40–47.

FRANCIS, R. J., ROSE, J., AND CHUNG, K. 1990. Chortle: A technology mapping program for lookup table-based field programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (Orlando, FL, June), 613–619.

FRANCIS, R. J., ROSE, J., AND VRANESIC, Z. G. 1991a. Technology mapping of lookup table-based FPGAs for performance. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 568–571.

FRANCIS, R. J., ROSE, J., AND VRANESIC, Z. G. 1991b. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 227–233.

FRIEDMAN, S. J. AND SUPOWIT, K. J. 1990. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput. 39,* 5, 710–713.

FUJITA, M. AND MATSUNAGA, Y. 1991. Multi-level logic minimization based on minimal support and its application to the minimization of look-up table type FPGAs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 560–563.

FUJITA, M. AND KUKIMOTO, Y. 1992. Patching method for lookup-table type FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Vienna, Aug.), 61–70.

GABOW, H. 1976. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. ACM 23,* (Apr.), 221–234.

GAO, T., CHEN, K.-C., CONG, J., DING, Y., AND LIU, C. L. 1993. Placement and placement-driven technology mapping for FPGA synthesis. In *Proceedings of the IEEE International ASIC Conference* (Rochester, NY, Sept.), 91–94.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco.

GROH, M. 1991. Technology mapping for look-up table FPGAs. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Oxford, England, Sept.), 191–200.

HALATSIS, C. AND GAITANIS, N. 1978. Irredundant normal forms and minimal dependence sets of a Boolean function. *IEEE Trans. Comput. 27,* 11, 1064–1068.

HE, J. AND ROSE, J. 1994. Technology mapping for heterogeneous FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

HE, S. AND TORKELSON, M. 1993. Decomposition of logic functions with partial vertex chart. In *Proceedings of the IEEE International ASIC Conference* (Rochester, NY, Sept.), 430–433.

HEAP, M. A., ROGERS, W. A., AND MERCER, M. R. 1992. A synthesis algorithm for two-level XOR based circuits. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 459–462.

HU, A. J. AND DILL, D. L. 1993. Reducing BDD size by exploiting functional dependencies. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 266–271.

HUANG, J.-D., JOU, J.-Y., AND SHEN, W.-Z. 1995. Compatible class encoding in Roth-Karp decomposition for two-output LUT architecture. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 359–363.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE 40,* 9, 1098–1101.

HWANG, T.-T., OWENS, R. M., AND IRWIN, M. J. 1992. Efficiently computing communication complexity for multilevel logic synthesis. *IEEE Trans. Comput. Aided Des. 11,* 5, 545–554.

HWANG, T.-T., OWENS, R. M., IRWIN, M. J., AND WANG, K.-H. 1994. Logic synthesis for field-programmable gate arrays. *IEEE Trans. Comput. Aided Des. 13,* 10, 1280–1287.

JOHNSON, D. S., DEMERS, A., ULLMAN, J. D., GAREY, M. R., AND GRAHAM, R. L. 1974. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput. 3,* 299–325.

KAPOOR, B. 1994. An efficient graph-based technology mapping algorithm for FPGAs using lookup tables. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

KARP, R. M. 1963. Functional decomposition and switching circuit design. *J. SIAM 11,* 2, 291–335.

KARPLUS, K. 1991. Xmap: A technology mapper for table-lookup field-programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 240–243.

KARPLUS, K. 1989. Using if-then-else DAGs for multi-level logic minimization. In *Proceedings of the Decennial Caltech Conference on VLSI* (Pasadena, CA, March), 101–118.

KERNIGHAN, B. W. AND LIN, S. 1970. An efficient heuristic procedure for partitioning of electrical circuits. *Bell Syst. Tech. J. 49,* 2, 291–308.

KEUTZER, K. 1987. DAGON: Technology binding and local optimization by DAG matching. In *Proceedings of the ACM/IEEE Design Automation Conference* (Miami Beach, FL, June), 341–347.

KIRKPATRICK, S., GELAT, C. D., AND VECCHI, M. P., JR. 1983. Optimization by simulated annealing. *Science 220,* (May), 671–680.

KOMMU, V. AND POMERANZ, I. 1993. GAFPGA: Genetic algorithm for FPGA technology mapping. In *Proceedings of the European Design Automation Conference* (Hamburg, Germany, Sept.), 300–305.

KUKIMOTO, Y. AND FUJITA, M. 1992. Rectification method for lookup-table type FPGA's. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 54–61.

LAI, Y.-T. AND SASTRY, S. 1992. Edge-valued binary diagrams for multi-level hierarchical verification. In *Proceedings of the ACM/IEEE Design Automation Conference* (Anaheim, CA), 608–613.

LAI, Y.-T., PAN, K.-R. R., AND PEDRAM, M. 1994. FPGA synthesis using function decomposition. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 30–35.

LAI, Y.-T., PAN, K.-R. R., PEDRAM, M., AND VRUDHULA, S. 1993b. FGMap: A technology mapping algorithm for lookup table type FPGAs based on function graphs. In *Proceedings of the International Workshop on Logic Synthesis* (Tahoe City, CA, May) 9b.1–4.

LAI, Y.-T., PEDRAM, M., AND VRUDHULA, S. 1993a. BDD based decomposition of logic functions with application to FPGA synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 642–647.

LAM, W. K. C. AND BRAYTON, R. K. 1992. On relationship between ITE and BDD. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 448–451.

LAWLER, E. L., LEVITT, K. N., AND TURNER, J. 1969. Module clustering to minimize delay in digital networks. *IEEE Trans. Comput. 18,* 1, 47–57.

LEGL, C., WURTH, B., AND ECKL, K. 1996. An implicit algorithm for support minimization during functional decomposition. In *Proceedings of the European Design and Test Conference* (Paris, March).

LEVIN, I. AND PINTER, R. Y. 1993. Realizing expression graphs using table-lookup FPGAs. In *Proceedings of the European Design Automation Conference* (Hamburg, Germany, Sept.), 306–311.

LU, A., SAUL, J., AND DAGLESS, E. 1994. Architecture oriented logic optimization for lookup table based FPGAs. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 26–29.

MADRE, J. C. AND BILLON, J. P. 1988. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the ACM/IEEE Design Automation Conference* (Anaheim, CA), 205–210.

MALIK, S., SENTOVICH, E. M., AND BRAYTON, R. K. 1991. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Trans. Comput. Aided Des. 10,* 1, 74–84.

MATHUR, A. AND LIU, C. L. 1994. Performance driven technology mapping for lookup-table based FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

MATSUNAGA, Y. AND FUJITA, M. 1989. Multi-level logic minimization using binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 556–559.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1995. *Logic Synthesis for Field-Programming Gate Arrays.* Kluwer, Norwell, MA.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1994. Optimum functional decomposition using encoding. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Diego, CA, June), 408–413.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1993a. Some results on the complexity of Boolean functions for table look up architectures. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 505–512.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1993b. Sequential synthesis for table look up programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 224–229.

MURGAI, R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1992. An improved synthesis algorithm for multiplexor-based PGA's. In *Proceedings of the ACM/IEEE Design Automation Conference* (Anaheim, CA, June), 380–386.

MURGAI, R., NISHIZAKI, Y., SHENOY, N., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1990. Logic synthesis algorithms for programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (Orlando, FL, June), 620–625.

MURGAI, R., SHENOY, N., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1991a. Performance directed synthesis for table look up programmable gate arrays. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 572–575.

MURGAI, R., SHENOY, N., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1991b. Improved logic synthesis algorithms for table look up architectures. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 564–567.

MUROGA, S., KAMBAYASHI, Y., LAI, H. C., AND CULLINEY, J. N. 1989. The transduction method—design of logic networks based on permissible functions. *IEEE Trans. Comput. 38,* 10, 1404–1424.

PAN, P. AND LIU, C. L. 1996. Optimal clock period FPGA technology mapping for sequential circuits. In *Proceedings of the ACM/IEEE Design Automation Conference* (Las Vegas, NV, June), 720–725.

PANDA, S., SOMENZI, F., AND PLESSIER, B. F. 1994. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 628–631.

PAPADIMITRIOU, C. H. AND STEIGLITZ, K. 1982. *Combinatorial Optimization: Algorithm and Complexity.* Prentice-Hall, Englewood Cliffs, NJ.

ROSE, J., EL GAMAL, A., AND SANGIOVANNI-VINCENTELLI, A. 1993. Architectures of field-programmable gate arrays. *Proc. IEEE 81,* 7, 1013–1029.

ROTH, J. P. AND KARP, R. M. 1962. Minimization over Boolean graphs. *IBM J. Res. Dev.* (Apr.) 227–238.

RUDELL, R. 1993. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 42–47.

SANGIOVANNI-VINCENTELLI, A., EL GAMAL, A., AND ROSE, J. 1993. Synthesis methods for field programmable gate arrays. *Proc. IEEE 81,* 7, 1057–1083.

SASAO, T. 1993. FPGA design by generalized functional decomposition. In *Logic Synthesis and Optimization,* Ed. Sasao, T., Norwell, MA (Jan.), 233–257.

SAUCIER, G., BRASEN, D., AND HIOL, J. P. 1993a. Partitioning with cone structures. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Santa Clara, CA, Nov.), 236–239.

SAUCIER, G., FRON, J., AND ABOUZEID, P. 1993b. Lexicographical expressions of Boolean functions with application to multilevel synthesis. *IEEE Trans. Comput. Aided Des. 12,* 11, 1642–1654.

SAUL, J. 1991. An algorithm for the multi-level minimization of Reed-Muller representations. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 634–637.

SAWADA, H., SUYAMA, T., AND NAGOYA, A. 1995. Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 353–358.

SAWKAR, P. AND THOMAS, D. 1993. Performance directed technology mapping for look-up table based FPGAs. In *Proceedings of the ACM/IEEE Design Automation Conference* (Dallas, TX, June), 208–212.

SAWKAR, P. AND THOMAS, D. 1992. Area and delay mapping for table-look-up based field programmable gate arrays. In *Proceedings of the ACM/IEEE Design Automation Conference* (Anaheim, CA, June), 368–373.

SCHAFER, I. AND PERKOWSKI, M. A. 1993. Synthesis of multiplexer circuits for incompletely specified multioutput Boolean functions with mapping to multiplexer based FPGAs. *IEEE Trans. Comput. Aided Des. 12,* 11, 1655–1664.

SCHLAG, M., CHAN, P. K., AND KONG, J. 1991. Empirical evaluation of multilevel logic minimization tools for a field programmable gate array technology. In *Proceedings of the International Workshop on Field Programmable Logic and Applications* (Oxford, England, Sept.), 201–213.

SCHLAG, M., KONG, J., AND CHAN, P. K. 1994. Routability-driven technology mapping for lookup table-based FPGAs. *IEEE Trans. Comput.-Aided Des. 13,* 1, 13–26.

SCHLAG, M., KONG, J., AND CHAN, P. K. 1992. Routability-driven technology mapping for lookup table-based FPGAs. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA, Oct.), 86–90.

SCHUBERT, E., KEBSCHULL, U., AND ROSENSTIEL, W. 1994. Functional decision diagrams for technology mapping to lookup-table FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

SHEN, W.-Z., HUANG, J.-D., AND CHAO, S.-M. 1995. Lambda set selection in Roth-Karp decomposition for LUT-based FPGA technology mapping. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 65–69.

SHIN, H. AND KIM, C. 1995. Performance-oriented technology mapping for LUT-based FPGAs. *IEEE Trans. VLSI Syst. 3,* 2, 323–327.

SOE, S. AND KARPLUS, K. 1993. Variable ordering heuristics for ordered binary decision diagrams and canonical if-then-else DAGs. In *Proceedings of the International Workshop on Logic Synthesis* (Tahoe City, CA, May), P3d.1–15.

STANION, T. AND SECHEN, C. 1995. A method for finding good Ashenhurst decomposition and its application to FPGA synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 60–64.

THAKUR, S. AND WONG, D. F. 1995. Simultaneous area and delay minimum K-LUT mapping for K-exact networks. In *Proceedings of the IEEE International Conference on Computer Design* (Austin, TX).

THAKUR, S., WONG, D. F., KRISHNAMOORTHY, S., AND MOCEYUNAS, P. 1995. Delay minimal decomposition of multiplexers in technology mapping. In *Proceedings of the International Workshop on Logic Synthesis* (Tahoe City, CA, May), 1.59–1.68.

TOGAWA, N., SATO, M., AND OHTSUKI, T. 1994. Maple: A simultaneous technology mapping, placement and global routing algorithm for field-programmable gate arrays. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 156–163.

TOUATI, H., SHENOY, N., AND SANGIOVANNI-VINCENTELLI, A. 1992. Retiming for table-lookup field-programmable gate arrays. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.), 89–93.

TREVILLYAN, L. 1993. An experiment in technology mapping for FPGAs using a fixed library. In *Proceedings of the International Workshop on Logic Synthesis* (Tahoe City, CA, May), P9c.1–6.

TRIMBERGER, S. M. 1994. *Field-Programmable Gate Array Technology.* Kluwer, Norwell, MA.

TRIMBERGER, S. M. 1993. A reprogrammable gate array and applications. *Proc. IEEE 81,* 7, 1030–1041.

WAN, W. AND PERKOWSKI, M. A. 1992. A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping. In *Proceedings of the European Design Automation Conference* (Hamburg, Germany, Sept.), 230–235.

WANG, A. R. 1989. Algorithms for multi-level logic optimization. UC Berkeley Tech. Memor. UCB/ERL M89/50 (Apr.).

WEINMANN, U. AND ROSENSTIEL, W. 1994. Logic module independent mapping for table-lookup FPGAs. In *Proceedings of the ACM/SIGDA International Workshop on Field Programmable Gate Arrays* (Berkeley, CA, Feb.).

WEINMANN, U. AND ROSENSTIEL, W. 1993. Technology mapping for sequential circuits based on retiming techniques. In *Proceedings of the European Design Automation Conference* (Hamburg, Germany, Sept.), 318–323.

WOO, N.-S. 1991. A heuristic method for FPGA technology mapping based on the edge visibility. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 248–251.

WURTH, B., ECKL, K., AND ANTREICH, K. 1995. Functional multiple-output decomposition: Theory and an implicit algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference* (San Francisco, CA, June), 54–59.

XILINX. 1994. *The Programmable Logic Data Book*. Xilinx, Inc., San Jose, CA.

YANG, H. AND WONG, D. F. 1994. Edge-map: Optimal performance driven technology mapping for iterative LUT based FPGA designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (San Jose, CA, Nov.), 150–155.

Received December 1995; revised February 1996; accepted March 1996