where M is the set of all nodes IM_j excluding those fulfilling property 3. The larger the cost of a vector v is, the less test vectors can be produced by the grouping if v is selected.

The cost of a fault f, with V_f denoting the set of its test vectors, is then estimated as

$$\mathrm{FaultCost}(f) = \sum_{v \in V_f} \mathrm{WeightVector}(v).$$

A larger fault cost indicates that it is harder for the algorithm to cover it.

REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1990.

[2] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudo-Random Techniques*. New York: Wiley, 1987.

[3] V. Agrawal, C. Kime, and K. Saluja, "A tutorial on built-in self-test part 1: Principles," *IEEE Design Test Computers*, pp. 73–82, Mar. 1993.

[4] H.-J. Wunderlich, "BIST for systems-on-a-chip," *Integration, VLSI J.*, vol. 26, no. 1-2, pp. 55–78, Dec. 1998.

[5] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing*: Kluwer, 2000.

[6] K.-T. Chen and C.-J. Lin, "Timing driven test point insertion for full-scan and partial-scan BIST," in *Proc. Int. Test Conf.*, 1995, pp. 506–514.

[7] A. Stroele and H.-J. Wunderlich, "TESTCHIP: A chip for weighted random pattern generation, evaluation, and test control," *IEEE J. Solid-State Circuits*, vol. 26, pp. 1056–1063, July 1991.

[8] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Trans. Comput.*, vol. 44, pp. 223–233, Feb. 1995.

[9] H.-J. Wunderlich and G. Kiefer, "Bit-flipping BIST," in *Proc. Int. Conf. Computer-Aided Design*, 1996, pp. 337–343.

[10] S. Hellebrand, H.-G. Liang, and H.-J. Wunderlich, "A mixed mode BIST scheme based on reseeding of folding counters," in *Proc. Int. Test Conf.*, 2000, pp. 778–784.

[11] N. A. Touba and E. J. McCluskey, "Bit-fixing in pseudorandom sequences for scan BIST," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 545–555, Apr. 2001.

[12] K. Chakrabarty, B. T. Murray, and V. Iyengar, "Built-in test pattern generation for high-performance circuits using twisted-ring counters," in *Proc. IEEE VLSI Test Symp.*, 1999, pp. 22–27.

[13] N. A. Touba and E. J. McCluskey, "Test point insertion based on path tracing," in *Proc. VLSI Test Symp.*, 1996, pp. 2–8.

[14] J. Hartmann and G. Kemnitz, "How to do weighted random testing for BIST," in *Proc. Int. Conf. Computer-Aided Design*, 1993, pp. 568–571.

[15] C. Okmen, M. Keim, R. Krieger, and B. Becker, "On optimizing BIST-architecture by using OBDD-based approaches and genetic algorithms," in *Proc. VLSI Test Symp.*, 1997, pp. 426–431.

[16] C.-A. Chen and S. K. Gupta, "A methodology to design efficient BIST test pattern generators," in *Proc. Int. Test Conf.*, 1995, pp. 814–823.

[17] ——, "Efficient BIST TPG design and test set compaction via input reduction," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 692–705, Aug. 1998.

[18] K. Chakrabarty, B. Murray, J. Liu, and M. Zhu, "Test width compression for built-in self test," in *Proc. Int. Test. Conf.*, 1997, pp. 327–337.

[19] I. Hamzaoglu and J. Patel, "Reducing test application time for built-in-self-test test pattern generators," in *Proc. VLSI Test Symp.*, 2000, pp. 369–375.

[20] N. A. Touba and E. J. McCluskey, "Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST," in *Proc. Int. Test Conf.*, 1995, pp. 674–682.

[21] M. F. Alshaibi and C. R. Kime, "MFBIST: A BIST method for random pattern resistant circuits," in *Proc. Int. Test Conf.*, 1996, pp. 176–185.

[22] C. Fagot, P. Girard, and C. Landrault, "On using machine learning for logic BIST," in *Proc. Int. Test Conf.*, 1997, pp. 338–346.

[23] L. R. Huang, J. Y. Jou, and S. Y. Kuo, "Gauss-elimination-based generation of multiple seed-polynomial pairs for LFSR," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1015–1024, Sept. 1997.

[24] S. Chiusano, P. Prinetto, and H. J. Wunderlich, "Non-intrusive BIST for systems-on-a-chip," in *Proc. Int. Test Conf.*, 2000, pp. 644–651.

BDS: A BDD-Based Logic Optimization System

Congguang Yang and Maciej Ciesielski

***Abstract*—This paper describes a novel logic decomposition theory and a practical logic synthesis system, *BDS*. It is based on a new binary decision diagrams (BDD) decomposition technique which supports all types of decomposition structures, including AND, OR, XOR, and complex MUX, both algebraic and Boolean. As a result, the method is very efficient in synthesizing both AND/OR and XOR-intensive functions. It also has a capability to handle very large circuits, as it employs the BDD decomposition in the partitioned Boolean network environment. The experimental results show that BDD-based logic decomposition is a promising alternative to the existing logic optimization approaches. In particular, it offers a superior runtime advantage over traditional logic synthesis systems.**

***Index Terms*—BDD, logic optimization, synthesis.**

I. INTRODUCTION

Traditional logic optimization methodology, based on algebraic factorization [1], [2], has gained tremendous success and emerged as a dominant method in logic synthesis. However, while near optimal results can be obtained for AND/OR-intensive functions of control and random logic, results are far from satisfactory for arithmetic and XOR-intensive logic functions, which can be more compactly represented as a combination of AND/OR and XOR expressions. Although logic optimization methods based on Boolean factorization can potentially offer better results than algebraic methods, they failed to compete with algebraic techniques due to their high computational complexity. We believe that this failure of Boolean optimization techniques is caused by inappropriate data structure used to represent Boolean functions. The predominant cube representation used by those techniques naturally favors algebraic-based methods and is not suitable for Boolean operations. Consequently, Boolean operations such as MUX and XOR received less attention from the onset of logic synthesis research.

We believe that logic synthesis methods will keep evolving with the emergence of newer and more efficient logic representations, and in particular with the accumulation of expertise in binary decision diagrams (BDDs). This paper presents the first results of research that address this new opportunity. It presents a novel theory and a set of efficient techniques for logic decomposition based on BDD representation. We show that logic optimization can be efficiently carried out through an iterative BDD decomposition and manipulation. Our approach proves to be very efficient for both AND/OR- and XOR-intensive functions. To the best of our knowledge, this is the first unified logic optimization methodology that allows one to optimize such diverse classes of logic functions. We also present a practical and complete BDD-based logic optimization system, *BDS*, that can handle arbitrarily large circuits. It employs the BDD decomposition techniques in the partitioned Boolean network environment.

Manuscript received July 13, 2001. This work was supported in part by the National Science Foundation under Contract CCR-9901254. This paper was recommended by Associate Editor E. Macii.

C. Yang is with Chameleon Systems, Inc., San Jose, CA 95134 USA (e-mail: cyang@chameleonsystems.com).

M. Ciesielski is with the Department of Electrical and Computer Engineering, University of Massachusetts at Amherst, Amherst, MA 01003-4410 USA (e-mail: ciesiel@ecs.umass.edu).

Publisher Item Identifier S 0278-0070(02)05630-0.

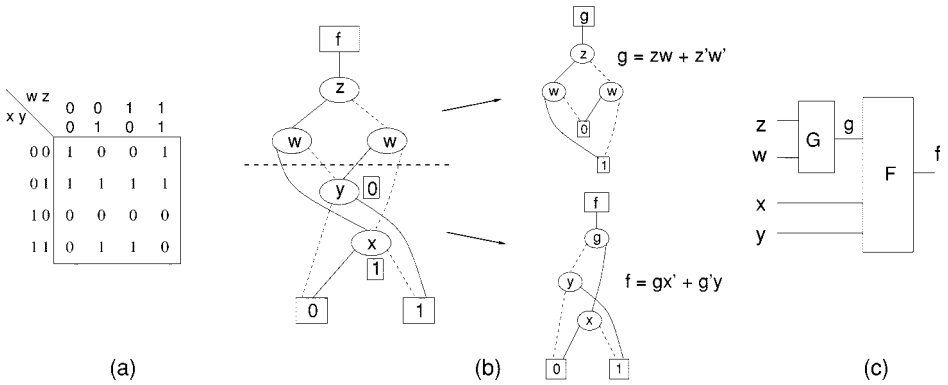

Fig. 1. Ashenhurst decomposition using BDD: (a) decomposition chart, (b) disjoint decomposition of the BDD, and (c) block diagram.

II. Background and Previous Work

A. Boolean Functions and BDDs

It is assumed that the reader is familiar with basic concepts of Boolean functions, Boolean networks, and BDDs. This section reviews basic terms used throughout the paper.

A *completely specified* Boolean function with n-inputs and one output is a mapping $f: B^n \longrightarrow B$, where $B = \{0, 1\}$. Such a function can be uniquely defined by its *onset*, $\mathrm{ON}(f) = \{x: f(x) = 1\}$, and its *offset*, $\mathrm{OFF}(f) = \{x: f(x) = 0\}$. For completely specified Boolean functions f and g, f covers g, denoted as $f \supseteq g$, if $\mathrm{ON}(f) \supseteq \mathrm{ON}(g)$. An *incompletely specified* Boolean function with n inputs and one output is a mapping $ff: B^n \longrightarrow Y$, where $Y = \{0, 1, *\}$, and $*$ stands for *don't care*. The *don't care set* (dc-set) of an incompletely specified Boolean function $f(x)$ is defined as $DC(f) = \{x: f(x) = *\}$. A cover F of an incompletely specified Boolean function f satisfies the condition $\mathrm{ON}(f) \subseteq F \subseteq \mathrm{ON}(f) \cup DC(f)$. The *support* of Boolean function F, denoted $\mathrm{supp}(F)$, is defined as the set of variables on which F depends. In the context of this work, we are only concerned with completely specified Boolean functions. In the sequel, the term *Boolean function* is used for a completely specified Boolean function.

The concept of BDDs was first proposed by Lee [3] in 1959. It was then developed into a useful data structure by Akers [4] and subsequently by Bryant [5], who introduced a concept of *reduced, ordered* BDDs (*ROBDDs*), along with a set of efficient operators for their manipulation, and proved the canonicity property of ROBDDs. The size of a BDD can be further reduced by introducing *complement edges*, [4], [6]. Basically, a complement edge (*c-edge*), points to the complementary form of the function (BDD node). To maintain canonicity, it is assumed that a complement edge can only be assigned to the 0-edge. In the rest of the paper, BDD refers to a reduced ordered BDD (ROBDD). In the drawings, the positive cofactor will be represented by a solid *1-edge*, and the negative cofactor by a dashed *0-edge*.

Multilevel circuits are typically represented as a *Boolean network*, a directed acyclic graph (DAG) whose nodes represent Boolean functions. Various Boolean network presentations differ mainly in the way they represent *local functions*, pertaining to individual nodes. The functionality of a Boolean node can be represented as a set of product terms (as in SIS [2]), or as a BDD, in a form known as a *local BDD representation*. A Boolean network can be also represented in a *global* form, by collapsing the entire Boolean network into a single node for each primary output. In this representation each global node is represented as a single monolithic BDD.

B. Functional Decomposition

The first systematic approach to functional decomposition was proposed by Ashenhurst [7] and Curtis [8]. According to this decomposition, a Boolean function $f(X)$ can be expressed as: $f(X) = F(G_1(Y), G_2(Y), \ldots, G_k(Y), Z)$, where $Y \cup Z = X$. Here Y is referred to as a *bound set* and Z is a *free set*. The original Ashenhurst decomposition calls for the two sets to be disjoint, $Y \cap Z = \emptyset$ (the *disjoint* decomposition) [1] and having a single predecessor block G, with $k = 1$ (called the *simple* decomposition); see Fig. 1(c). Under such a decomposition Boolean function can be represented by a *decomposition chart*, with the variables in Y and Z corresponding to the column and row indexes, respectively, as shown in Fig. 1(a). A disjoint decomposition $f(X) = F(G(Y), Z)$ exists if the number of distinct columns of the decomposition chart, or *column multiplicity*, is $\mu = 2$. Roth and Karp [9] extended this result to nondisjoint decomposition, with $Y \cap Z = X$, and with $k > 1$ predecessor blocks G_i. The functional decomposition methods based on the decomposition charts are computationally inefficient because the number of columns in the chart grows exponentially with the number of bound set variables, and testing decomposition with each bound set is possible only after constructing its decomposition chart.

C. Previous Work in BDD Decomposition

This section reviews a number of BDD-based logic decomposition methods developed over the last decade. These methods can be divided into two major classes: 1) methods that follow the traditional functional decomposition of Ashenhurst–Curtis, but rely on BDDs as an efficient data structure for the implementation of their algorithms and 2) methods that use the *structure* of a BDD to identify good decompositions and more efficiently utilize the expressive power of BDDs. The method described in this paper belongs to the latter category.

The BDD decomposition methods from the first class employ BDDs as a platform to carry out traditional functional decomposition of Ashenhurst [7] and Roth–Karp [9]. Lai *et al.* [10] demonstrated that the structure of a BDD is implicitly related to the decomposition chart and hence can be used to perform the functional decomposition. Given an ordered BDD, a *cut set* is selected that partitions the variables into a *bound set* and a *free set*; see Fig. 1(b). Each node in the cut set corresponds to a unique column of the decomposition chart, Fig. 1(a). The decomposition exists if the size of the cut set m satisfies the condition $m \leq 2^k$, where k is the number of outputs of function block G. The implementation of the decomposed functions, F and G, is accomplished by encoding the BDD nodes in the cut set, as shown in Fig. 1(b). The described cut-based approach has served as a basis for several logic decomposition methods [10]–[13]. They are particularly applicable to FPGA designs; in this case the cut is selected based on the number of inputs to the look-up-table (LUT) blocks. The

[1]The disjoint and nondisjoint decompositions refer to the interaction of the variables in the support of the function. They should not be confused with disjunctive (OR) and conjunctive (AND) decompositions, described in Section III-B.

method of Lai *et al.* [10] has been extended also to the decomposition of multiple-output functions in [14]. Here the multiple-output Boolean function is first converted into an integer-valued function and represented as an *edge-valued BDD* (EVBDD); the EVBDD is then decomposed using method similar to [10] and the result converted back into a multiple-output function. A serious limitation of all those methods is that they require finding a cut which separates the bound variables from the free variables.

An important group of methods in the same category is that of *bidecompositions*, introduced by Bochman *et al.* [15]. Bidecompositions are functional decompositions of the type $F(X,Y,Z) = G(X,Y) \odot H(Y,Z)$, where $\odot$ stands for any binary Boolean operation. For purely *algebraic* decomposition, $Y = \emptyset$, and for *nonalgebraic* decomposition $Y \neq \emptyset$ (the overlap set). If the support of G (or H) is identical to that of F, the bidecomposition is called *weak*; otherwise, it is called *strong*.

A class of *quasialgebraic decomposition*, i.e., bidecompositions where the set Y is fixed, has been introduced by Stanion and Sechen in [16]. They give the necessary and sufficient conditions for a function to have a quasi-algebraic decomposition for a given choice of X, Y, Z. This is a special case of Roth–Karp decomposition with $k = 1$ predecessor block. This method also requires that the variable partitioning into subsets X, Y, Z be consistent with the ordering of variables in the BDD.

The BDD-based logic synthesis continues to be an active research area. Recently, Mishchenko *et al.* [17] suggested a method to perform bidecomposition using formulas with quantifiers evaluated with the help of BDDs. A work of Files and Perkowski [18] applies multivalued decision diagrams, MDDs, to perform multivalued functional decomposition.

The second class of methods relies on the *structure* of BDD to identify good decompositions and guide directly the decomposition process. The first known work in this class, and the one that inspired our research, is that of Karplus [19]. Karplus introduced the concept of a *1-* and *0-dominator* and showed their relationship to *algebraic* AND/OR decomposition, illustrated in Fig. 2. In a BDD without the complement edges, a 1-dominator (0-dominator) is a node which belongs to every path from the root to terminal node 1 (0).

There have been several other attempts to perform multilevel logic optimization directly on a BDD. Bertacco and Damiani [20] proposed a method which performs recursive decomposition directly on a BDD. Their method basically annotates disjoint decomposition inherent in the BDD structure. It is fast and for some circuits generates much better results than SIS [2]. However, it can only detect simple disjoint decompositions. Stanion and Sechen [21] proposed a Boolean division and factorization method using a specialized BDD operator, called *interval cofactor*. An important contribution of this work is its capability to extract XORs using a BDD decomposition technique similar to that described in Section III-D. However, due to a lack of efficient way to generate good Boolean divisors, the improvement offered by such a Boolean division over SIS is marginal.

It should be noted that the BDD-based decomposition techniques mentioned here can only detect bidecompositions for a variable order consistent with their partitioning into the bound set and the free set. Otherwise, none of these methods can detect algebraic or quasi-algebraic decomposition and require reordering of variables. The method described in this paper attempts to remedy this problem. We demonstrate that the structure of a well-ordered BDD can be used *directly* to identify functional decomposition of the underlying function, leading to efficient multilevel logic implementations. The described method can detect algebraic as well as Boolean decompositions even for the variable order that is inconsistent with the variable partitioning. The theory of such a decomposition is the subject of Section III.

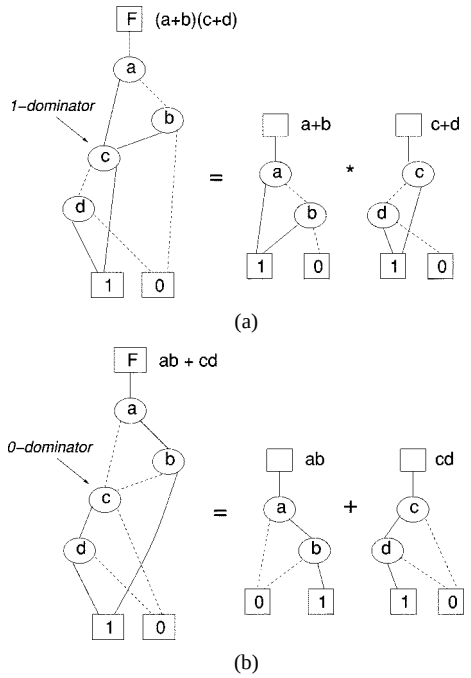

Fig. 2. Algebraic decompositions of Karplus: (a) conjunction decomposition, $F = (a+b)(c+d)$, based on 1-dominator and (b) disjunctive decomposition, $F = ab + cd$, based on 0-dominator.

III. Theory of BDD Decomposition

A. Terminology and Fundamentals

Definition 1 (BDD): A BDD is a DAG representing a Boolean function. It can be uniquely defined as a tuple, $BDD = (\Phi, V, E, \{0,1\})$, where Φ is the function node (*root*), V is the set of nodes, E is a set of edges, and 0 and 1 are the terminal nodes. □

Definition 2 (Leaf Edges): The *leaf edge* is an edge $e \in E$ which is directly connected to a terminal node of the BDD. The set of leaf edges, denoted Σ, can be partitioned into Σ_0, the set of leaf edges connected to 0, and Σ_1, the set of leaf edges connected to 1. All the other edges of the BDD are called *internal*. □

Definition 3 (Paths): A path from root to terminal node 0(1) is called a *0-path* (*1-path*). Π_0 is the set of all 0-paths, and Π_1 is the set of all 1-paths. $\Pi = \Pi_0 \bigcup \Pi_1$ is the set of all paths of the BDD. □

Theorem 1: Every internal edge $e \in (E - \Sigma)$ belongs to at least one path $p_1 \in \Pi_1$ and to one path $p_0 \in \Pi_0$.

Proof: The theorem is proved by contradiction. Since BDD is a connected graph, every edge must belong to Π_0 or Π_1. Assume that $e \in E - \Sigma$ belongs to Π_1 only. Then all the nodes below e can be collapsed into 1, so that $e \in \Sigma_1$. Hence, the contradiction. Same reasoning applies to the case of Π_0. □

Definition 4 (Cut): The *cut* in a BDD is a set of edges which partitions its nodes V into two disjoint subsets, D and $(V\text{-}D)$, such that *root* $\in D$ and terminals $0, 1 \in (V\text{-}D)$). A *horizontal cut* is a cut in which the support of D and $(V\text{-}D)$ in terms of the associated variables, are disjoint. □

We now provide a theoretical basis for two fundamental BDD decompositions, namely the conjunctive and disjunctive Boolean decompositions.

Definition 5 (Conjunctive Decomposition): Boolean function F has the *conjunctive (AND) decomposition* if it can be represented as $F = D \cdot Q$. Function D is called the Boolean *divisor* and Q is the *quotient* of F under this decomposition. □

Definition 6 (Disjunctive Decomposition) : Boolean function F has the *disjunctive (OR) decomposition* if it can be represented as $F = G + H$. □

The decomposition is *algebraic* if the supports of Q and D (G and H) are disjoint; otherwise, the decomposition is *Boolean*. In contrast to quasi-algebraic methods [16], we do not make any assumption whether the decomposition is algebraic or Boolean; nor do we require any explicit declaration of the number of the overlapping variables. Instead, we develop a general decomposition method for the conjunctive (disjunctive) decomposition, common to both types.

We now state two well-known theorems that form a basis of our BDD decomposition technique. The proofs can be found in any textbook on logic synthesis, such as [22].

Theorem 2: Boolean function F has a conjunctive Boolean decomposition $F = D \cdot Q$ if and only if $F \subseteq D$. For a given choice of D, the quotient Q must satisfy $F \subseteq Q \subseteq F + \overline{D}$.

We use this theorem to generate the Boolean divisor and the quotient directly from the BDD. Our procedure will first generate divisor D and then compute quotient Q from F using the offset of D as don't care.

Theorem 3: Boolean function F has a disjunctive Boolean decomposition $F = G + H$ if and only if $F \supseteq G$. Then, for a given choice of G, the disjunctive term H must satisfy the condition $\overline{F} \subseteq \overline{H} \subseteq \overline{F} + G$.

This theorem forms the basis for our disjunctive decomposition; we will first generate G and then compute $\overline{H}$ from $\overline{F}$ using an onset of G as don't care.

B. And/Or Decomposition

Definition 7 (Generalized Dominator) : Consider a cut partitioning the set of BDD nodes of function F into D and (V-V_D). The portion of the BDD defined by nodes V_D is copied to form a separate graph, where an edge e is connected to 0 if $e \in \Sigma_0(F)$, and it is connected to 1 if $e \in \Sigma_1(F)$. All the internal edges $e \in (E - \Sigma)$ are left dangling; they are referred to as *free* edges. The resulting graph is called the *generalized dominator* of F with respect to the given cut, denoted $GD(F)$. □

Example 1: Fig. 3(a) and (b) shows the construction of a generalized dominator. First, a cut is applied to the BDD of F in Fig. 3(a). Then the portion above the cut is copied to form a separate graph, with $\Sigma(F)$ edges connected to the corresponding terminals 0 or 1, shown in Fig. 3(b). □

The following theorem shows how to obtain a Boolean divisor and perform the conjunctive Boolean decomposition of the BDD of F by redirecting the free edges of $GD(F)$ to terminal node 1.

Lemma 1: Given a generalized dominator $GD(F)$ of function F, the Boolean divisor D is obtained from $GD(F)$ by redirecting its free edges to 1. The quotient Q is obtained from F by redirecting the $\Sigma_0(D)$ edges in F to don't care nodes.

Proof: First we shall show that D satisfies the condition of Theorem 2 , that is $D \supseteq F$. By construction, $\Sigma_0(D) \subseteq \Sigma_0(F)$, and $\Pi_0(D) \subseteq \Pi_0(F)$, that is $\overline{D} \subseteq \overline{F}$, or equivalently, $D \supseteq F$. In Fig. 3, $\overline{D} = \{\bar{e}\bar{d}\} \subseteq \overline{F} = \{\bar{e}\bar{d},\ \bar{e}d\bar{b}\}$. Alternatively, it can be argued that all 1-paths of D are either identical to or subsume those of F. This is because, by Theorem 1, every internal edge of F is on some 1-path. By construction, this is also true for the free edges of $GD(F)$. By redirecting the free edges of $GD(F)$ to 1, the BDD of D covers all 1-paths of F, that is $D \supseteq F$.

It remains to be shown that, for such constructed D, the BDD of Q satisfies the condition for the quotient: $F \subseteq Q \subseteq F + \overline{D}$. This follows directly from the construction of Q; we start with $Q = F$, identify the offset of D as 0-paths in the BDD of D, and redirect the corresponding 0-paths in F to don't care (DC) nodes. By construction, each 0-path in D, $p_0(D)$, has an equivalent 0-path in F, $p_0(F)$; that is, the nodes of $p_0(D)$ are in one-to-one correspondence with the nodes of $p_0(F)$. For example, see the 0-path ($\bar{e}\bar{d}$) in the BDD of D in Fig. 3(d) and the identical one in F in Fig. 3(a). Replacing the node 0 with DC on such a path in $Q = F$ will never decrease the onset of the resulting Q compared to F [it may only increase it by making some nodes redundant, as in Fig. 3(e) and (f)]. Hence $Q \supseteq F$. Finally, we claim that redirecting the $\Sigma_0(D)$ edges to DC is equivalent to adding the offset of D as a don't care set to F. This is true, because replacing node 0 by DC at the end of the path $p_0(F)$ amounts to adding an offset cube of D associated with $p_0(D)$ to the BDD of F. This is done for all $p_0(D) \in \Pi_0(D)$. Hence such constructed Q satisfies $Q \subseteq F + \overline{D}$. □

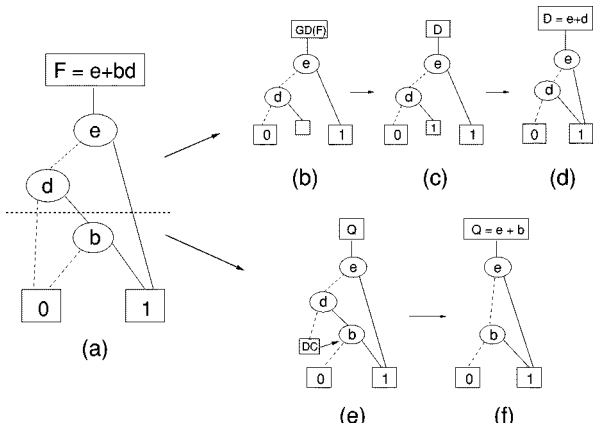

Fig. 3. A simple example of conjunctive Boolean decomposition.

Example 2: Fig. 3 is used again to illustrate Lemma 1 for function $F = e + bd$. The free edge (d) of $GD(F)$ in Fig. 3(b) is redirected to constant 1, as shown in Fig. 3(c). The Boolean divisor D is readily evaluated as $D = e + d$, shown in Fig. 3(d). The quotient Q for this divisor is obtained from F by using the offset of D (cube $\bar{e}\bar{d}$) as don't care; see Fig. 3(e). The minimization of F with respect to this don't care gives $Q = e + b$, as shown in Fig. 3(f). Notice that $(D = e + d) \supset (F = e + bd)$, and $(Q = e + b)$ satisfies the condition $F \subset Q \subseteq F + \bar{e}\bar{d}$.

Example 3: A complete conjunctive (AND) decomposition is shown in Fig. 4. First, a cut is performed on the BDD in Fig. 4(a), and the generalized-dominator $GD(F)$ is built. Then, the Boolean divisor D is constructed from $GD(F)$ by redirecting the free edges to 1. The reduction of this BDD gives $D = (af + b + c)$, shown in Fig. 4(b). The quotient Q is obtained from F by minimizing it with the offset of D (i.e., $\overline{D} = \{a\bar{f}\bar{b}\bar{c},\ \bar{a}\bar{b}\bar{c}\}$) as don't care, giving $Q = (ag + d + e)$; see Fig. 4(c). As a result, $F = (af + b + c)(ag + d + e)$ with only eight literals. This is the best know decomposition for this function. □

Disjunctive (OR) decomposition is dual to the conjunctive decomposition. The following is the fundamental theorem for disjunctive decomposition.

Lemma 2: Consider a disjunctive Boolean decomposition $F = G + H$. Given a generalized dominator $GD(F)$ of function F, the Boolean term G can be obtained by redirecting the free edges of $GD(F)$ to 0. The Boolean term H is obtained from F by redirecting the $\Sigma_1(G)$ edges in F to don't care nodes.

Proof: We must show that such constructed G and H satisfy the conditions of Theorem 3 . First, notice that by redirecting the free edges of $GD(F)$ to 0, the offset of the resulting Boolean function G covers the offset of F, that is, $\overline{G} \supseteq \overline{F}$, hence $G \subseteq F$. The rest of the proof is dual to that of Lemma 1 . In this case, each 1-path $p_1(G)$ has its counterpart in F; see for example the path $\{ab\}$ in Fig. 5(d) of D and the one in Fig. 5(a) of F. Redirecting the $\Sigma_1(G)$ edges to DC is equivalent to replacing the onset of G as don't care in F in the construction of H. □

Example 4: Fig. 5 illustrates Lemma 2 for function $F = ab + bc$. The free edges of $GD(F)$ in Fig. 5(b) are redirected to constant 0,

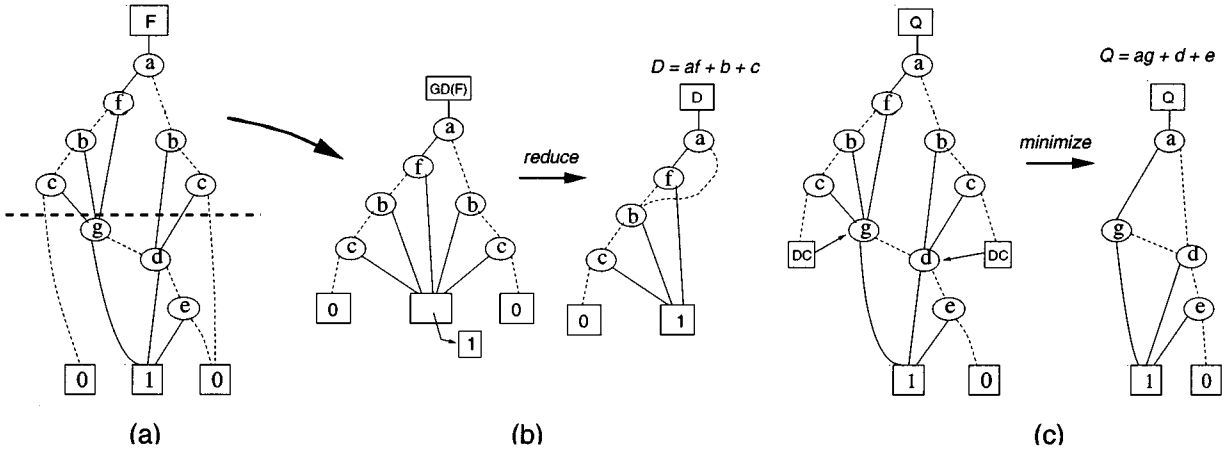

Fig. 4. Conjunctive BDD decomposition: (a) original function F, (b) generalized dominator and Boolean divisor D, and (c) computing quotient Q from F.

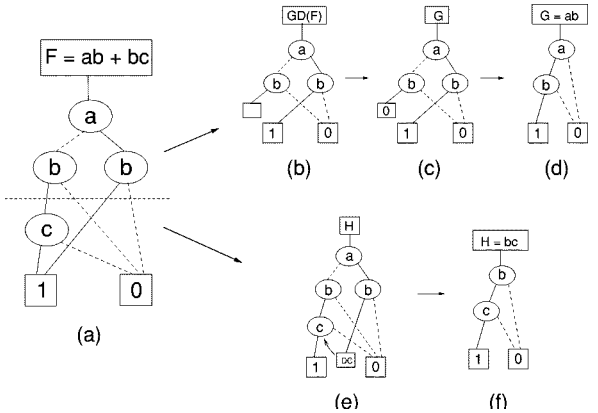

Fig. 5. A simple example of disjunctive Boolean decomposition.

resulting in $G = ab$, as shown in Fig. 5(c) and (d). The term H of this decomposition is obtained from F by setting the onset of G, $\{ab\}$, as don't care; see Fig. 5(e). The minimization of F with respect to this don't care gives $H = bc$, or $\overline{H} = \bar{b} + \bar{c}$, as shown in Fig. 5(f). Notice that $G = ab \subset F$, and $\overline{F} \subseteq \overline{H} \subset (\overline{F} + ab)$.

The conjunctive and disjunctive decompositions described in this section are, in general, *Boolean* decompositions. This is because, by construction, the functions D and Q (or G and H) share their support variables; while $\mathrm{supp}(D)$ contains only the variables above the cut, $\mathrm{supp}(Q) \subseteq supp(F)$ because Q is derived from F. That is, using the terminology of [15], the resulting bidecomposition may be weak. If the minimization of Q removes all variables in $\mathrm{supp}(D)$ from $\mathrm{supp}(Q)$, leading to disjoint supports of D and Q, the resulting decomposition is algebraic. In this case, our generalized dominator reduces to a 1- or 0-dominator of Karplus [19], discussed in Section II-C. The same argument applies to the disjunctive decomposition, $F = G + H$.

In contrast to quasi-algebraic methods of [16], our method can find decomposition for the variable order not necessarily consistent with the partitioning of variables into sets X, Y, Z. Consider, for example, function $F = (ab + c)(ad + e)$. This decomposition can be readily obtained with our method. The algorithm of [16] can find this decomposition only when variable a separates the sets (b, c) and (d, e) in the variable order. However, the best variable order, (a, b, c, d, e), which gives the BDD of minimum size (eight nodes), violates this condition, making it impossible to obtain this decomposition.

Finally, we should comment on the minimization of the BDD with don't cares which is an essential part of our decomposition procedures (refer to proofs of Lemma 1 and 2). This problem has been shown to be NP-complete [23], [24], and only a few heuristics are available today to solve it. The exact method described in [24] can only be used for small functions. We use the heuristics based on the RESTRICT operator of Coudert and Madre [25].

C. Identifying Useful Cuts

The number of possible cuts that should be examined in the search for an optimal AND/OR decomposition can be prohibitively large even for a moderately sized BDD. Therefore, a mechanism to reduce the number of candidate cuts has been developed, rendering some cuts invalid or redundant.

It can be shown that only cuts which contain at least one leaf edge $e \in \Sigma$ can lead to nontrivial Boolean decomposition [26]. We refer to them as *valid cuts*. All terminal edges of a generalized dominator generated from other cuts are free; when redirected to 1 (0), they create trivial Boolean divisors $(D = 1)$, or trivial disjunctive Boolean terms $(H = 0)$. To further limit the number of cuts, they can be grouped into equivalence classes as follows.

Definition 8 (Equivalent cuts): Two cuts are *0-equivalent* if they contain the same set of Σ_0 edges. Similarly, two cuts are *1-equivalent* if they contain the same set of Σ_1 edges. □

Theorem 4: All Boolean divisors of a conjunctive decomposition, obtained from 0-equivalent cuts, are identical. Similarly, all Boolean terms of a disjunctive decomposition, obtained from 1-equivalent cuts, are identical.

Proof: Consider two 0-equivalent cuts. In each of the Boolean divisors generated by those cuts, edges $e \in \Sigma_0$ are connected to 0; all other edges are connected to 1. Hence, both Boolean divisors have the same set of 1-paths Π_1 (onset) and the same 0-paths Π_0 (offset). Hence, they are identical. Similar argument applies to 1-equivalent cuts. □

Fig. 6(a) shows a BDD with several possible cuts. Cuts 2 and 3 are 0-equivalent, hence they lead to identical Boolean divisors, as illustrated in Fig. 6(b) and (c). Additional properties, such as *transitive cut property* [26], can further reduce the number of valid cuts to be considered. In our approach we limit our attention to *horizontal cuts*. While it is obvious that nonhorizontal cuts can help identify other useful decompositions, possibly leading to better results, the inclusion of those cuts would significantly increase the computational complexity. In the worst case, the total number of horizontal cuts is $|V|$, where V is the number of variables (levels of a BDD). In practice, the total number of valid horizontal cuts is much smaller because many cuts are either 1- or 0-equivalent.

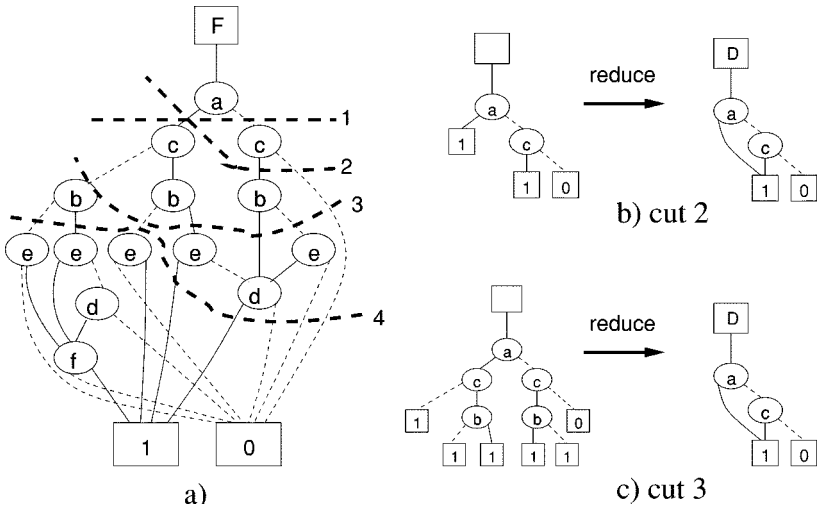

Fig. 6. (a) Various cuts on a BDD. (b), (c) Equivalent cuts.

D. XOR Decomposition

The BDD decomposition based on generalized dominators, described in the previous section, relies on the leaf edges, Σ. While BDDs of random logic AND/OR-intensive functions tend to have many Σ edges, XOR-intensive and arithmetic functions have very few or no Σ edges. It is apparent that the decomposition which relies on Σ edges will fail on a BDD with few Σ edges.

This section addresses this issue by developing the techniques targeting XOR-type decomposition on a BDD. We observed that an XOR decomposition is associated with the presence of *complement edges* (c-edges) in the BDD. For this reason, we use a BDD representation with c-edges to detect such decompositions. Recall that in order to maintain canonicity, only the negative edges can be complemented. They will be represented in this paper as dotted edges with a bubble. In the sequel, we will use XNOR ($\overline{\oplus}$) instead of XOR, as it is more straightforward to develop.

We shall first consider an *algebraic* XNOR decomposition, $F = G\overline{\oplus}H$, with disjoint supports of G and H. Let function F be represented by a BDD with complement edges. We define an *x-dominator* in such a BDD to help identify an algebraic XOR decomposition.

Definition 9 (x-Dominator): Node $v \in V$ which is contained in every path $p \in \Pi$ is called an *x-dominator*. □

The definition of *x-dominator* implies that there must exist at least one complement edge above it; otherwise, all the BDD nodes above v will collapse into v.

Theorem 5: Let v be an x-dominator in the BDD of Boolean function F. The BDD of F can be algebraically decomposed as $F = G\overline{\oplus}H$, where G is a BDD rooted at v; BDD of H is obtained from F by redirecting the regular edges pointing to node v to terminal 1 and the complement edges pointing to node v to terminal 0.

Proof: Fig. 7(a) shows a generic BDD with an x-dominator v. The BDD of G rooted at v is copied with negative polarity ($\overline{G}$) so that the complement edges pointing to G can be transformed into negative edges pointing to $\overline{G}$, as shown in Fig. 7(b). The BDD of F can now be represented as a disjunction of two parts, one with $\overline{G}$ replaced by node 0, and the other with G replaced by node 0, as shown in Fig. 7(c). Note that G and $\overline{G}$ are 1-dominators in their respective BDDs. By defining H to be a Boolean function derived from F by redirecting all the edges pointing to G to node 1, and all the edges pointing to $\overline{G}$ to 0, as in Fig. 7(c), function F can be represented as $F = G\cdot H + \overline{G}\cdot\overline{H} = G\overline{\oplus}H$. □

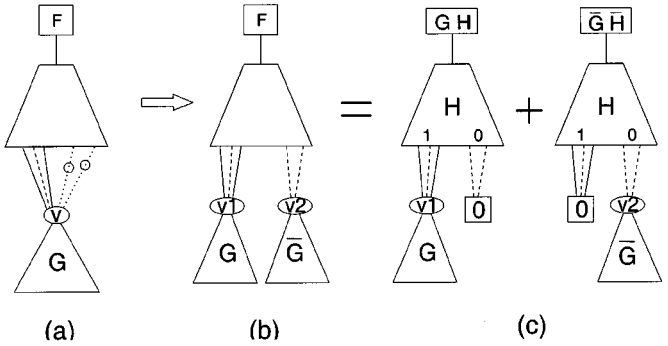

Fig. 7. Algebraic XNOR decomposition based on x-*dominator*.

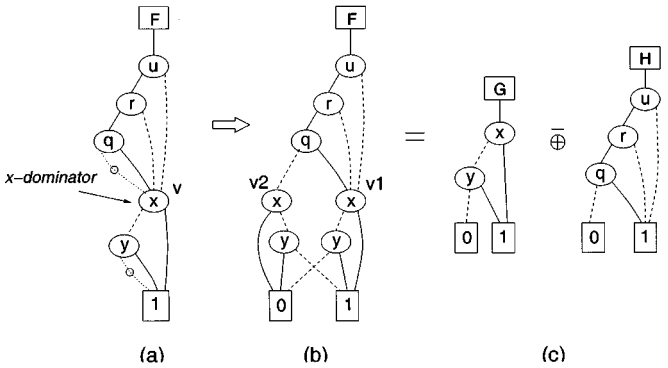

Fig. 8. An x-*dominator* leading to algebraic XNOR decomposition, $F = (x + y)\overline{\oplus}(\bar{u} + \bar{r} + q)$.

Notice that 1-paths of F that pass through node v_1 are always *complementary* with respect to (w.r.t.) those passing through node v_2, hence the portions of BDD above the two nodes are H and $\overline{H}$, respectively.

Example 5: Fig. 8(a) shows a BDD with c-edges for Boolean function $F = \{(\bar{u} + \bar{r} + q)(x + y) + ur\bar{q}\bar{x}\bar{y}\}$. An x-dominator can be identified at node v (variable x); the function rooted at v is $G = x + y$. By expressing the c-edge coming into v as a negative edge, the BDD can be represented as a BDD with regular edges in Fig. 8(b). Here node v is split into v_1, associated with $G = x + y$, and node v_2, associated with $\overline{G} = \bar{x}\bar{y}$. In this BDD all 1-paths pass either through node v_1 or through node v_2. Therefore, $F = (x + y)\overline{\oplus}(\bar{u} + \bar{r} + q)$, see Fig. 8(c). □

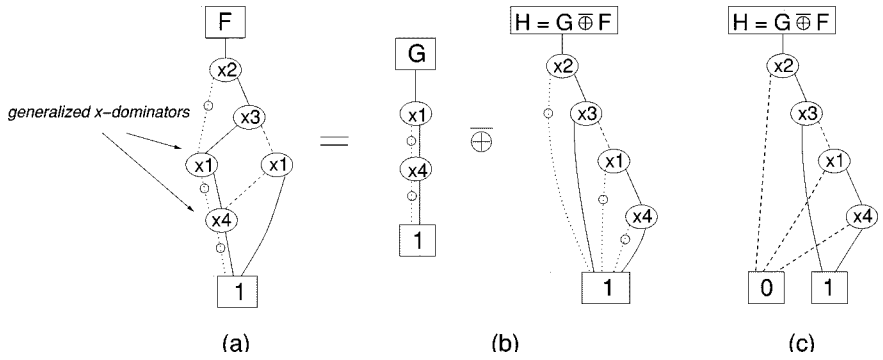

Fig. 9. Boolean XNOR decomposition of function *rnd4-1*, $F = (x_1 \overline{\oplus} x_4) \overline{\oplus} (x_2(x_3 + x_1 x_4))$.

The algebraic XOR decomposition defined here is essentially identical to that of [21].

We shall now consider *Boolean* XNOR decomposition, $F = G \overline{\oplus} H$, with no constraints imposed on the supports of G and H. While function F may not have an algebraic XOR decomposition, it has many Boolean XOR decompositions, as expressed by the following theorem.

Theorem 6: For a Boolean function F and an arbitrary Boolean function G, there always exists a Boolean function H, such that $F = G \overline{\oplus} H$.

Proof: The proof is based on the Boolean transformation: $F = G \overline{\oplus} (G \overline{\oplus} F) = G \overline{\oplus} H$, where G is an arbitrary Boolean function, and $H = G \overline{\oplus} F$. □

This theorem points out that a Boolean XOR decomposition is not unique, in fact it has infinitely many decompositions, each associated with an arbitrary Boolean function G. While exhaustive search for all combinations of G, H that minimize F is clearly prohibitive, a set of good candidates for G can be detected directly from the BDD by identifying a *generalized x-dominator*, defined below.

Definition 10 (Generalized x-dominator): Node $v \in V$ which is pointed to by at least one complement and one regular (positive or negative) edge is called the *generalized x-dominator*. □

Once a generalized x-dominator G is identified in the BDD, $H = G \overline{\oplus} F$ is computed using a standard apply operator from a BDD package.

Example 6: Fig. 9(a) shows the BDD for circuit rnd4–1 from the MCNC benchmark suite. There are two generalized x-dominators, namely x_1, and x_4. We illustrate an XNOR decomposition based on x_1. First we create $G = x_1 \overline{\oplus} x_4$, as a function rooted at x_1; see Fig. 9(b). The BDD of H is derived from G and F by computing $H = G \overline{\oplus} F$, as shown in Fig. 9(b). We show it also without the c-edges in Fig. 9(c) to point out that it exposes a 1-dominator x_3, so it can be further algebraically decomposed as $H = x_2(x_3 + x_1 x_4)$. This results in the final decomposition: $F = (x_1 \overline{\oplus} x_4) \overline{\oplus} (x_2(x_3 + x_1 x_4))$. □

E. Mux Decomposition

Each node of a BDD can be viewed as a multiplexor (MUX), leading to a *simple MUX decomposition*. Such a decomposition can be generalized to a more effective *functional MUX decomposition*, where the control signal is a function, instead of a single input variable. Such a decomposition often leads to concise multilevel implementations.

Theorem 7: Consider a BDD structure, in which two nodes, u and v, cover all paths $p \in \Pi$. The BDD can then be decomposed as $F = hf + \bar{h}g$, where f and g are functions rooted at nodes u and v, respectively, and h is obtained from the BDD of F by redirecting node u to 1, and node v to 0.

Proof: The proof is similar to that of Theorem 5; see Fig. 10. □

Example 7: Fig. 11 shows an example of a functional MUX decomposition for $F = (zw + \bar{z}\bar{w})\bar{x} + (z\bar{w} + \bar{z}w)x$, shown in Fig. 1(b).

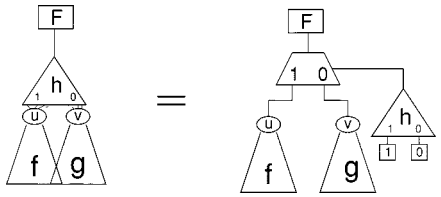

Fig. 10. Functional MUX decomposition: $F = hf + \bar{h}g$.

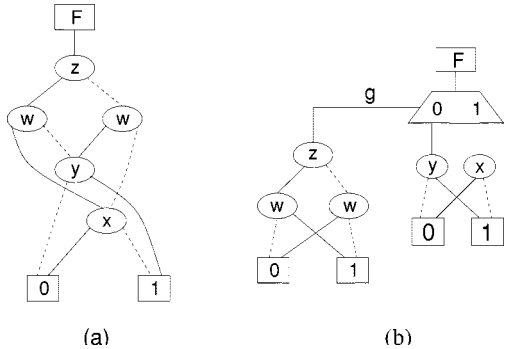

Fig. 11. Example of functional MUX decomposition: $F = g\bar{x} + \bar{g}y$, where $g = zw + \bar{z}\bar{w}$.

Two "articulation" nodes, x and y, of this BDD cover all paths $p \in \Pi$. Subsequently, the function can be represented as $F = g\bar{x} + \bar{g}y$, where $g = zw + \bar{z}\bar{w}$ serves as a control signal for the MUX. □

We should recall that Theorem 7 applies only to BDDs without complement edges above u and v. One should note the resemblance of the functional MUX decomposition with the functional decomposition of Ashenhurst (cf. Section II-C). Specifically, the MUX decomposition with a single control function is identical to a simple disjoint decomposition of Ashenhurst with column multiplicity of two; in general, the column multiplicity corresponds to the number of the "articulation" nodes (u, v) in Theorem 7, as illustrated by the above example.

IV. BDS System—Implementation

This section briefly reviews the implementation of a complete logic optimization system, BDS. In order to handle arbitrarily large circuits it operates in the *partitioned* Boolean network environment. The details of the initial implementation of the system are presented in [26] and [27].

A. Synthesis Flow

BDS adopts a general synthesis flow of SIS, as shown in Fig. 12. The similarity between BDS and SIS is obvious. The fundamental difference between the two systems is in the way they represent Boolean

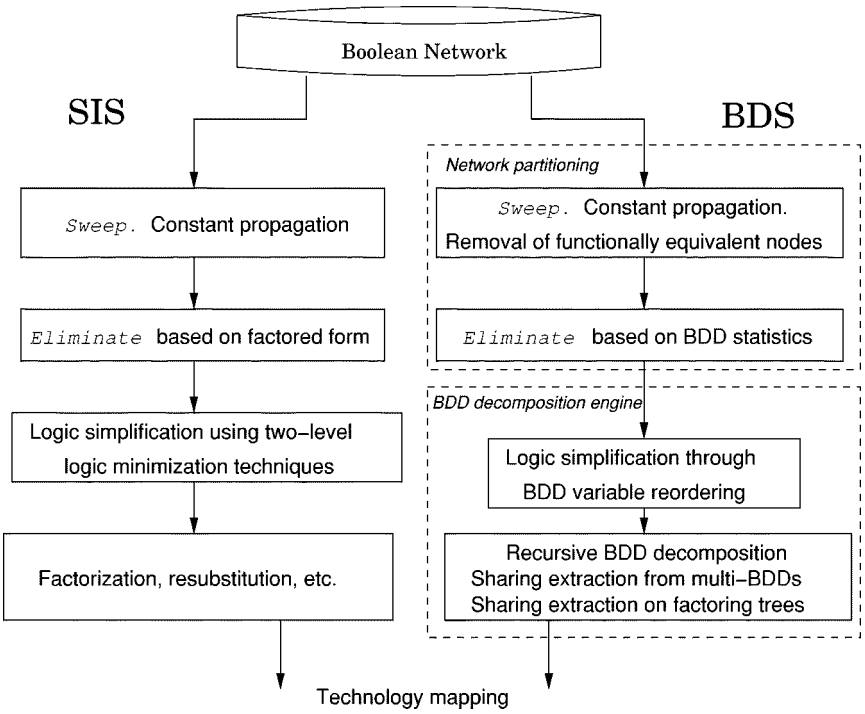

Fig. 12. Synthesis flows of SIS and BDS.

nodes and carry out the individual optimization procedures. SIS works on an algebraic representation of the entire Boolean network, iteratively factoring out algebraic expressions and performing node collapsing and logic simplification. BDS first partitions the network into a set of nodes, represents each as a *local BDD*, and then performs BDD decomposition. All the subsequent procedures are carried out on local BDDs, using the decomposition algorithms especially tailored for BDDs.

The first step in the employed synthesis flow is the removal of initial redundancy from the Boolean network using procedure sweep. While there is no real logic optimization involved in this procedure, it plays an important role in preparing the network for a subsequent decomposition. In addition to removing constant and single-variable nodes, all functionally equivalent nodes are also identified and removed from the Boolean network. Removal of functionally duplicated nodes at this initial stage significantly improves runtime complexity of *BDS* over traditional approaches.

B. Network Partitioning By Node Elimination

Applying logic optimization to the entire Boolean network using *global* BDD representation may not be practical for large designs. On the other hand, applying logic optimization to a completely *local* representation may not work either, as it may leave a significant amount of redundancy in the network. A reasonable tradeoff can be achieved by *partially* collapsing the Boolean network into a set of *supernodes*. Each supernode can then be represented as a local BDD and synthesized. Partial collapsing is critical to a logic synthesis system; it helps to remove logic redundancy, caused, for example, by local *reconvergence*, often present in a multilevel network.

Partial collapsing can be implemented with a help of the $\mathrm{eliminate}$ procedure, which attempts to maintain the right granularity of the Boolean network. A properly designed $\mathrm{eliminate}$ scheme provides a good starting point for logic optimization algorithms. Two approaches have been proposed in the literature for the $\mathrm{eliminate}$ procedure using BDDs. The first one is based on *progressive elimination* [28], where BDDs are constructed from primary inputs to primary outputs. At any point, if the size of a BDD is larger than a predefined fixed threshold, an intermediate variable is introduced. This approach, however, ignores a specific structure of the Boolean network. As a result, the elimination often stops at boundaries which are not natural for a given logic network; this approach may also cause memory explosion. The second approach is based on *iterative elimination* [29] and is quite similar to the $\mathrm{eliminate}$ procedure of SIS [2]. *BDS* adopts a similar approach, except that it uses the number of BDD nodes as the cost function to guide the elimination, instead of the literal count.

In practice, a straightforward implementation of the $\mathrm{eliminate}$ procedure is complicated by the BDD variable reordering. When local BDDs are constructed for a Boolean network, an intermediate variable is created for each Boolean node. Therefore, in addition to all primary inputs, a BDD manager also contains all intermediate variables. The number of such variables could be very large even for a medium-sized circuit, and reordering a BDD manager with all the variables will severely degrade the overall runtime performance. Furthermore, the removal of one node from the Boolean network corresponds to the demise of one variable in the BDD manager; such a variable becomes *unused*. After several iterations and the removal of many Boolean nodes the BDD manager contains a large number of unused variables. It has been found that in the entire ISCAS benchmark set about 63% of variables in the BDD manager become unused just after first iteration. Obviously, performing variable reordering in a BDD manager with such a large number of unused variables is highly inefficient. In our system, instead of reordering the BDD manager with all the variables, a new BDD manager, containing only the *used* variables, is initialized. Each BDD is then transferred into the new BDD manager using our proprietary bddPool mechanism, described in [27]. During this process, variables are substituted according to a mapping function $\mathcal{M}$, which maps the variables from the old BDD manager onto the new one. When all BDDs are reconstructed in the new BDD manager, a set of BDDs which are isomorphic to the original ones, but much more compact in the range of indexes, is obtained. This process is referred to as a *BDD mapping* [27]. Thanks to an efficient implementation of BDD mapping our $\mathrm{eliminate}$

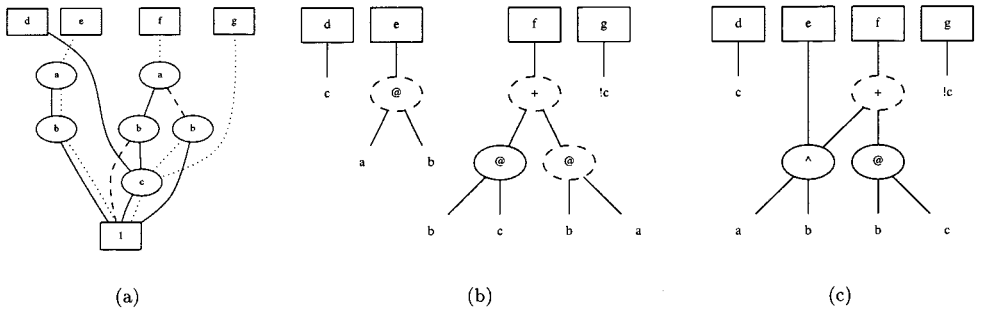

Fig. 13. Sharing extraction on the factoring trees: (a) original BDD, (b) factoring trees after BDD decomposition, and (c) factoring trees after sharing extraction ($\wedge$ = XOR; $\textcircled{=}$ = XNOR; dotted oval = complemented gate).

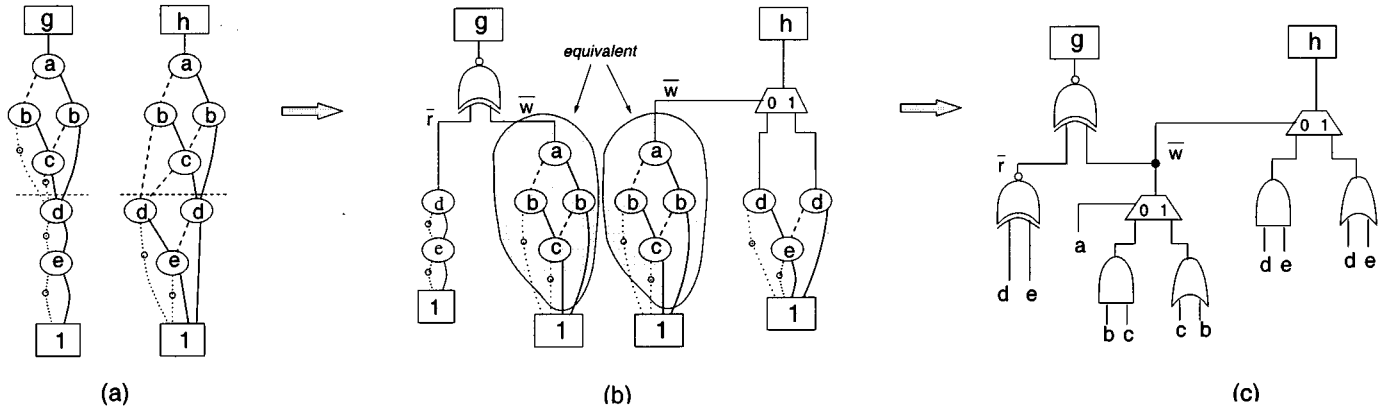

Fig. 14. Decomposition and extraction sharing on the factoring trees of g and h.

procedure is on average 85 times faster than that of [29]. The runtime advantage of *BDS* becomes even more pronounced for larger circuits.

C. The BDD Decomposition Engine

In our system, a BDD is first subjected to a variable reordering [30]. This serves as a means to achieve an initial logic simplification, a good starting point for further logic decomposition. The decomposition of the ordered BDD consists of two major parts: 1) an *iterative BDD decomposition*, where a large BDD is recursively decomposed into smaller parts, and 2) a construction and processing of the *factoring trees*. Factoring trees are constructed along with the BDD decomposition as a means to record the result of the decomposition.

The iterative BDD decomposition is a search process for the most efficient decomposition. The BDD dominators, introduced in Section III are empirically ordered in terms of the resulting decomposition efficiency as follows: 1) simple dominators (1-, 0- and x-dominator); 2) functional MUX; 3) generalized dominator; and 4) generalized x-dominator. If all searches fail, the BDD is decomposed using a simple cofactor (simple MUX) w.r.t. a top variable in the BDD. In practice, this last step is rarely reached; it is kept to ensure that the BDD will still be decomposed when all other attempts fail.

A BDD decomposition process begins with the BDD structural scan in which the structural information of a BDD needed to guide the various decomposition types is obtained. The result of BDD decomposition for each output is stored in a *factoring tree*. Subsequently, *logic sharing* between different factoring trees is detected to further optimize the synthesis results. For this purpose, BDDs are constructed for all factoring trees in a bottom-up fashion, and the canonicity property of a BDD is used to identify functionally equivalent subtrees. Fig. 13 shows an example of sharing extraction for circuit *b1.blif* from the MCNC benchmark set.

Example 8: Consider a two-output function, $\{g, h\}$, with global BDD representations shown in Fig. 14(a). The BDDs are decomposed independently of each other, one using x-dominator and the other using functional MUX decomposition. The factoring trees are constructed for each function, resulting in the structure shown in Fig. 14(b). Finally, the logic shared between the two factoring trees is extracted and shared in the final network, as shown in Fig. 14(c). □

Finally, we should comment on difference between our approach to the the decomposition of multiple-output functions and the one based on EVBDDs, described in [14]. An EVBDD-based method requires that all outputs share the same bound set to be decomposable. Our approach, which decomposed each output independently, offers freedom to select each "bound set" individually, potentially leading to better results.

V. Experimental Results

The experiments have been conducted on a Pentium-III/500 machine running Linux. They cover all nontrivial combinational circuits from the MCNC benchmark set. The test circuits were divided into two groups: 1) AND/OR-intensive (random logic) functions, and 2) XOR-intensive, arithmetic functions. All the circuits were synthesized by both BDS and by SIS using *script.rugged* [2] and mapped onto *mcnc.genlib*. Both a tree-based mapper of SIS and a mapper based on Boolean matching, *ceres* [31], were used in the experiment.

The results for small and medium size circuits, which can be modeled as a global BDD, were presented and analyzed in [32]. For AND/OR-intensive (random logic) circuits, BDS uses on average 4% fewer gates but requires 5% more area than SIS. At the same time, BDS outperforms SIS by 37% in CPU time for this class of circuits. The slight increase in area is due to the higher cost of XOR gates assigned by the mapper. For the class of arithmetic functions and XOR-intensive logic, BDS outperforms SIS in all aspects: the number of literals (on average by 40%), gate count (by 23%), gate area (by 14%), and CPU time (by 84%). A tree-based SIS mapper was used

TABLE I
COMPARISON BETWEEN BDS AND SIS

Circuits	SIS				BDS			
	Area λ^2	Delay [ns]	CPU [s]	Mem [MB]	Area λ^2	Delay [ns]	CPU [s]	Mem [MB]
C1355	689	39.40	6.6	3.3	711	45.60	0.4	1.0
C1908	695	68.60	8.1	3.1	730	65.00	0.8	1.0
C3540	1695	81.40	16.1	15.1	1713	81.20	3.6	1.9
C432	290	75.90	46.1	6.4	357	78.40	0.2	0.5
C499	689	39.40	6.8	3.5	708	43.60	0.6	0.5
C5315	2286	68.60	10.2	5.6	2402	70.50	5.3	3.0
C6288	4631	237.8	21.8	14.8	4677	178.3	3.8	1.1
C7552	3038	115.70	54.2	45.2	3112	83.30	4.2	4.8
C880	567	56.10	1.9	2.2	563	43.20	0.7	0.8
pair	2274	74.30	16.1	6.8	2466	52.60	2.1	2.0
rot	965	51.60	4.5	2.7	1025	51.90	1.0	0.9
dalu	1306	61.0	70.5	4.8	2604	117.2	7.2	2.6
vda	837	39.8	19.7	3.3	1054	47.8	7.1	1.4
Total	19962	1009.6	282.6	116.8	22122	958.6	37.0	21.5

TABLE II
RESULTS OF BDS AND SIS FOR LARGE ARITHMETIC CIRCUITS

Circuits	SIS				BDS				Speedup
name	gates	area [λ^2]	delay [ns]	CPU [s]	gates	area [λ^2]	delay [ns]	CPU [s]	
bshift16	158	406.0	19.0	3.9	145	376.0	21.8	1.0	3.9
bshift32	292	774.0	27.5	19.1	255	704.0	31.1	2.3	8.3
bshift64	653	1796.0	34.9	100.2	570	1656.0	47.2	6.5	15.4
bshift128	1478	4237.0	55.5	643.9	1193	3750.0	75.3	22.9	28.1
bshift256	3683	9981.0	95.3	8666.4	2782	8614.0	132.6	28.9	300.0
bshift512	-	-	-	> 15 hrs	7367	22598.0	240.0	95.1	> 560.0
m2x2	8	17.0	9.1	0.2	11	22.0	5.7	0.1	2.0
m4x4	97	220.0	56.1	2.7	112	256.0	37.5	0.4	6.7
m8x8	514	1224.0	121.2	42.4	561	1351.0	81.8	2.2	19.3
m16x16	2312	5678.0	264.0	110.8	2517	6111.0	186.5	9.7	11.4
m32x32	9941	24213.0	531.3	1215.4	10511	25787.0	387.9	48.0	25.3
m64x64	41040	99787.0	1069.8	23881.7	42947	105749.0	789.3	321.8	74.2
Total	60176	148333	2073.7	34719.2	61604	154376	1796.7	443.8	

in this experiment since *ceres* was not stable on this set of circuits. As a result only 33% of XORs were preserved by the mapper. As demonstrated in [32], the performance of BDS in terms of the number of gates compares favorably with the technique of Tsai *et al.* [33], developed specifically for arithmetic functions.

Table I summarizes a set of larger experimental results from the *LGSynth91* test case suite, showing the circuit delay and memory usage by both systems. The circuits were mapped by the SIS mapper. The gate area of circuits synthesized by BDS is consistently larger than SIS, in this set by about 11% on average. The delay is on average 6% smaller than that of SIS. The amount of memory required by BDS is on average 82% lower. In terms of the CPU performance, BDS demonstrates significant advantage over SIS; on average it is more than eight times faster on this set of circuits. The results for two test cases, *dalu* and *vda*, merit additional explanation; they are inferior to SIS both in circuit area and delay. This can be explained by the fact that BDS does not perform node simplification with local and satisfiability don't cares derived from the network, as it is done in *full simplify* of SIS. A specialized BDD-based Boolean network optimization with don't cares would be a desirable feature in order to improve these results.

To prove the potential of BDS to optimize large circuits, where network partitioning into local BDDs is necessary, we tested our system on a set of arithmetic circuits generated by a proprietary HDL-to-blif translator. The results are shown in Table II. On average, BDS is over 100 times faster than SIS. The overall runtime complexity of BDS is significantly lower than that of SIS.

Notice that circuit area synthesized with *BDS* is only slightly (on average 3%) larger than that obtained with SIS. There are two reasons for that. First, since *BDS* has a capability to perform XOR and MUX decompositions, the XOR and MUX structures are represented *explicitly* in the factoring trees and in the final *blif* files. However, only a small fraction of XORs and MUXs are actually mapped to XOR and MUX gates; this is a known weakness of the tree-based technology mapper of SIS used in our experiment. Secondly, currently BDS does not have the capability to perform *satisfiability don't care* minimization. If the redundancy cannot be removed by the eliminate eliminate procedure, it will most likely remain in the final synthesized circuit.

All the results produced by BDS, except for C6288, were independently verified w.r.t. the original specification by our internal verifier (BDS with option *-verify*) and by SIS. Since both tools build global BDDs to perform verification, they could not verify the C6288 multiplier. However, since we verify each step of the elimination process (when building local BDDs) we believe the result to be correct too.

VI. CONCLUSION

The experimental results show that BDD-based logic optimization is a promising alternative to the existing logic optimization approaches. In

particular, it offers a superior runtime advantage over traditional logic synthesis techniques based on algebraic transformations. It can also be useful as a tool for fast and reliable estimation of logic optimization. An up-to-date version of the BDS software can be downloaded from [34].

The capability of current BDD-based methodology can be further enhanced by incorporating the following future work.

1) While BDS offers great runtime improvement, especially for arithmetic circuits, it cannot successfully compete in terms of gate area with highly tuned and perfected algebraic methods for random logic circuits. BDD-based logic minimization with satisfiability don't cares, similar to *full-simplify* of SIS, should be developed to improve the area performance of BDS.
2) The minimization of BDDs with don't care nodes (Section III-B) remains an open and difficult problem requiring more research. Improving this procedure could significantly improve the results.
3) One of the current weaknesses of BDS is its inability to properly balance the factoring tree, which is crucial for the delay minimization. This can be overcome by selecting, among several candidate dominators, the ones closest to the middle of the tree. This requires further tuning of the cost function.
4) Recently, we found that BDS is also amenable to FPGA synthesis. In-depth analysis of the underlying algorithms for BDD decomposition should be performed to fully understand the reason for its applicability to FPGAs. Very encouraging initial results, showing over 30% improvement in the LUT count, have already been obtained [35].
5) The common logic extraction performed on the factored trees is currently limited to completely specified functions of the tree nodes. The caching technique recently proposed in [17] can be readily used used to remedy this problem.

Compared with the state-of-the-art logic synthesis methodology, which has evolved from continuous research and development during the past 20 years, the presented BDD-based logic optimization technique is very young and much less mature. Extensive research must be performed to make this approach a truly successful synthesis method. We hope that this work will initiate a new round of research in logic synthesis area in the years to come.

ACKNOWLEDGMENT

The authors are indebted to A. Mishchenko for illuminating discussions on the BDD-based logic decomposition methods and for his help in providing a reliable methodology to verify the synthesis results. They would also like to thank the reviewers for providing insightful comments about the paper.

REFERENCES

[1] R. K. Brayton, G. D. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, pp. 264–300, Feb 1990.
[2] E. Sentovich *et al.*, "SIS: A System for Sequential Circuit Synthesis," ERL, Dept. EECS, Univ. California, Berkeley, UCB/ERL M92/41, 1992.
[3] C. Y. Lee, "Representation of switching circuits by binary decision programs," *Bell System Techn. J.*, vol. 38, no. 4, pp. 985–999, June 1959.
[4] S. B. Akers, "Functional testing with binary decision diagrams," in *Eighth Annual Conf. Fault-Tolerant Computing*, 1978, pp. 75–82.
[5] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, pp. 677–691, Aug. 1986.
[6] K. Brace, R. Rudell, and R. Bryant, "Efficient implementation of a BDD package," in *Proc. Design Automation Conf.*, 1990, pp. 40–45.
[7] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching*, vol. XXIX, Ann. Computation Lab. Harvard Univ., Cambridge, MA, 1959, pp. 74–116.
[8] H. A. Curtis, *A New Approach to the Design of Switching Circuits*. Boston, MA: D. Van Nostrand, 1962.
[9] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM J. Res. Dev.*, pp. 227–238, Apr. 1962.
[10] Y-T. Lai, M. Pedram, and S. Vrudhula, "Bdd based decomposition of logic for functions with applications to FPGA synthesis," in *Proc. Design Automation Conf.*, 1993, pp. 642–647.
[11] T. Sasao, *FPGA Design by Generalized Functional Decomposition, in Logic Synthesis and Optimization*. Boston, MA: Kluwer, 1993.
[12] Y.-T. Lai, K.-R. Pan, and M. Pedram, "OBDD-based function decomposition: Algorithms and implementattion," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 977–990, Aug. 1996.
[13] S.-C. Chang, M. Marek-Sadowska, and T. Hwang, "Technology mapping for TLI FPGA's based on decomposition of binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1226–1235, Oct. 1996.
[14] Y.-T. Lai, M. Pedram, and S. Vrudhula, "Evbdd-based algorithms for integer linear programming, spectracl transformation, and function decomposition," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 959–974, Aug. 1994.
[15] D. Bochman, F. Dresig, and B. Steinbach, "A new decomposition method for multilevel circuit design," in *Proc. Eur. DAC*, 1991, pp. 374–377.
[16] T. Stanion and C. Sechen, "Quasialgebraic decomposition of switching functions," in *Advanced Res. VLSI*, 1995.
[17] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proc. Design Automation Conf.*, 2001, pp. 103–108.
[18] C. Files and M. Perkowski, "New multi-valued functional decomposition algorithms based on MDD's," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1081–1086, Sept. 2000.
[19] K. Karplus, "Using if-then-else DAG's for multi-level logic minimization," Univ. California, Santa Cruz, UCSC-CRL-88-29, 1988.
[20] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," in *IEEE Int. Conf. Computer-Aided Design*, 1997, pp. 78–82.
[21] T. Stanion and C. Sechen, "Boolean division and factorization using binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1179–1184, Sept. 1994.
[22] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Boston, MA: Kluwer, 1996.
[23] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size for incompletely specified functions," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1435–1437, Nov. 1996.
[24] A. L. Oliveira, L. Carloni, T. Villa, and A. L. Sangiovanni-Vincentelli, "Exact minimization of binary decision diagrams using implicit techniques," *IEEE Trans. Computers*, vol. 47, pp. 1282–1296, Nov. 1998.
[25] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *Proc. ICCAD*, 1990, pp. 126–129.
[26] C. Yang and M. Ciesielski, "BDS: BDD-Based logic optimization system," Dept. Electrical and Computer Engineering, Univ. Massachusetts Amherst, TR-CSE-00-01, 2000.
[27] ——, "BDS: A BDD-based logic optimization system," in *Proc. Design Automation Conf.*, 2000, pp. 92–97.
[28] P. Buch, A. Narayan, R. Newton, and A. Sangiovanni-Vincentelli, "On synthesizing pass transistor logic," in *Intl. Workshop Logic Synthesis*, 1997.
[29] R. Chaudhry, T. Liu, A. Aziz, and J. Burns, "Area-oriented synthesis for pass-transistor logic," in *Int. Conf. Computer Design*, 1998, pp. 160–167.
[30] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *IEEE Int. Conf. Computer-Aided Design*, 1993, pp. 42–47.
[31] F. Mailhot and G. D. Micheli, "Algorithms for technology mapping based on binary decision diagrams and on boolean operations," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 599–620, May 1993.
[32] C. Yang, V. Singhal, and M. Ciesielski, "BDD decomposition for efficient logic synthesis," in *Int. Conf. Computer Design*, 1999, pp. 626–631.
[33] C. Tsai and M. Marek-Sadowska, "Multilevel logic synthesis for arithmetic functions," in *Proc. Design Automation Conf.*, 1996, pp. 242–247.
[34] BDS system [Online]. Available: http://www.ecs.umass.edu/ece/labs/vlsicad/ciesielski.html
[35] N. Vemuri, "BDD-Based logic synthesis for LUT-based FPGA's," Masters, Dept. Electrical Computer Engineering, Univ. Massachusetts Amherst, 2001.