BDD-BASED LOGIC OPTIMIZATION SYSTEM

Congguang Yang   Maciej Ciesielski

Feburary 2000

TR-CSE-00-1

cyang,ciesiel@ecs.umass.edu

Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003

BDD-Based Logic Optimization System

Congguang Yang Maciej Ciesielski
{cyang,ciesiel}@ecs.umass.edu
Department of Electrical & Computer Engineering
University of Massachusetts

I. Introduction

Logic synthesis plays a central role in the design automation of VLSI circuits. Software tools for logic synthesis are one of the most important tools ever developed in the area of Computer-Aided Design (CAD). With the help of those tools, a designer is freed from tedious and error-prone low-level circuit design, and can focus on architectural and algorithmic level issues.

Logic synthesis is composed of three main steps. First, a circuit described in high-level language (hardware description languages, such as VHDL or Verilog) is transformed into a Boolean network. Then, the Boolean network is optimized using *logic optimization* tools. Finally, the optimized Boolean network is mapped to a library of logic cells. The entire process is directed in such a way as to optimize certain design objectives (such as delay, area, power, etc) and meet users' specifications and constraints. Among these three steps, logic optimization is the most important. Because the quality of final synthesis results is mainly determined by it. As a result, intensive research has been done in this area.

A. Traditional Multi-Level Logic Optimization

The main theme in multi-level logic optimization is *factorization*. In a typical logic synthesis environment, a Boolean function is initially represented as a *sum-of-product* (SOP) or cube form. This form is transformed by factoring out common algebraic or Boolean expressions. In an algebraic factorization, logic functions are treated as polynomials, in which rules of Boolean algebra are not applied. Boolean factorizations, based on Boolean division, apply Boolean algebra rules, hence can produce better results in terms of the resulting logic complexity (number of terms, literals, etc).

Traditional logic optimization methodology, based on algebraic factorization for Boolean networks [1], [2], has gained tremendous success in logic optimization and emerged as the dominant method. However, while near optimal results can be obtained for those Boolean functions which can be represented with AND/OR expressions, results are far from satisfactory for functions which can be compactly represented as a combination of AND/OR and XOR expressions.

Although logic optimization methods based on Boolean factorizations, can potentially offer better results than algebraic methods, they failed to compete with algebraic methods due to their high computational complexity. We believe that the failure of Boolean optimization is caused by inappropriate data structure used to represent Boolean functions. *Cube* representation, which is derived from two-level AND/OR form (PLA), naturally favors algebraic-based methods. This representation, however, is not suitable for Boolean operations. Consequently, Boolean operations such as MUX and XOR received less attention from the beginning of logic synthesis research.

B. New Opportunity

Through the continuously intensive research and development in logic synthesis area for the last twenty years, the general framework for logic synthesis has been well established. While the space for further improvement of the synthesis flow seems to be limited, there is still potential for significant improvement in many procedures in a synthesis process [3]. This is especially true when more efficient ways to represent Boolean functions become available.

A brief review of logic synthesis history is shown in Fig. 1. It can be roughly divided into three periods, represented by three most famous methods: Quine-McCluskey and ESPRESSO for two-level logic minimization, and SIS for multi-level logic optimization. Quine-McCluskey method requires a Boolean function to be represented in the *minterm* form. Since the size of *minterm* representation is exponential in the number of inputs, this method is of theoretical importance only. ESPRESSO [4], the first practical logic minimization tool, works on the *sum-of-product* (SOP) form which is much more compact than *minterm*-based representation. The synthesis method in this category was later pushed to the limit by Coudert [5] by incorporating *implicit* enumeration techniques. Finally, SIS [2] is the most successful synthesis tool developed so far. It forms the backbone of most modern academic and commercial logic synthesis tools. The central theme in SIS is algebraic factorization in which *factored form*[1] was used as a Boolean logic representation. Compared with the SOP form, *factored form* is much more concise and closer to the final gate-level implementation.

This work has been supported by a grant from NSF under contract No. MIP-9613864.

[1] SIS still depends on two-level forms to carry out logic minimization of individual nodes of a Boolean network.

The history of logic synthesis demonstrates a simple, yet clear fact that the Boolean logic representation plays a central role in the evolution of synthesis methods. It seems quite natural that logic synthesis methods will keep evolving with the emergence of newer and more efficient Boolean logic representations. We believe that the pace of this evolution is increasing with the accumulation of expertise in Binary Decision Diagrams (BDDs). Our research is trying to address this new opportunity.

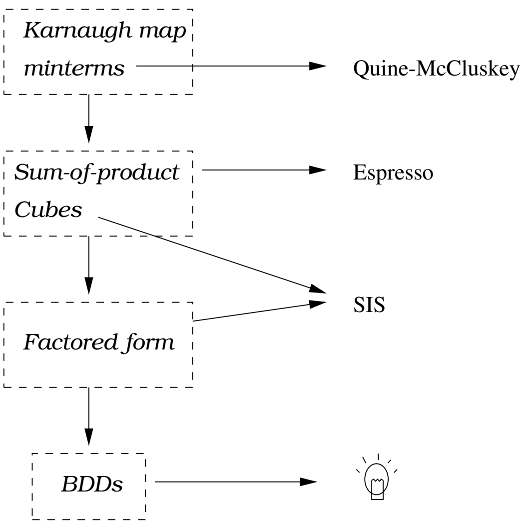


Fig. 1. A brief history of logic synthesis.

C. Main Contribution

A new BDD decomposition theory is presented in this paper. We show that *logic optimization* can be efficiently carried out through iterative BDD decomposition and manipulation. Our approach proves to be efficient for both AND/OR- and XOR-intensive functions. This is the *first* unified logic optimization methodology that allows to optimize both classes of functions.

We also propose a practical, complete, BDD-based logic optimization system, *BDS*, that can handle arbitrarily large circuits. A general framework which incorporates a typical logic synthesis procedures has been implemented in *BDS*. A number of new BDD manipulation techniques, which proved very efficient at manipulating BDDs in the partitioned Boolean network environment, are also presented.

II. Background and Terminology

A. Boolean Function

A *completely specified* Boolean function with n-inputs and 1 output is a mapping $f : \mathbf{B}^n \longrightarrow \mathbf{B}$, where $\mathbf{B} = \{0, 1\}$. A completely specified Boolean function can be uniquely defined by its *on-set* X^{on}, defined as $X^{on} = \{x : f(x) = 1\}$. Or *off-set*, defined as $X^{off} = \{x : f(x) = 0\}$. For completely specified Boolean functions f and g, f is covered by g, if $X^{on}_f \subseteq X^{on}_g$.

An *incompletely specified* Boolean function with n-inputs and 1 output is a mapping $ff : \mathbf{B}^n \longrightarrow \mathbf{Y}$, where $\mathbf{Y} = \{0, 1, *\}$, where * stands for *don't care*. The *don't care set* (dc-set) of an incompletely specified Boolean Boolean function $f(x)$ is defined as $X^{dc} = \{x : f(x) = *\}$. An incompletely specified Boolean function can be uniquely defined by its set (X^{on}, X^{dc}), (X^{off}, X^{dc}), or (X^{on}, X^{off}).

In the context of this work, we are only concerned with completely specified Boolean functions. In the sequel, a *Boolean function* is referred to as a completely specified Boolean function.

A.1 Representation of Boolean Functions

A Boolean function can be represented in many different *forms*. A form π is said to be *canonical* if the representation of a Boolean function *f* by π is *unique*.

An expression representing a Boolean function can be derived from its *truth table* by finding the sum of *rows* (*terms*) for which the function assumes value 1. The expression based on the sum of minterms is also referred to as *canonical sum-of-product* form.

Minterms are commonly used to represent Boolean functions. However, due to the exponential nature of this representations, which requires 2^n terms for a n-input function, its application is limited to simple Boolean functions, and mainly used for illustration purposes only.

A more practical representation of a Boolean function is the *sum-of-product* (SOP) form, which can be obtained by simplifying the minterm-based representation using rules of Boolean algebra. Each term in a SOP form is referred to as a *SOP term* (or a *product term*). Practically the number of product terms required to represent a function is much smaller than the number of minterms. However, because the simplification is not unique, the SOP form is not canonical.

In multi-level logic synthesis, a product term is also called a *cube*, and a SOP representation is referred to as a *set* of *cubes*. Formally, a *cube* is a product of literals, where a literal is a variable or its complement. *Cube* representation forms the backbone of all the logic synthesis systems.

However, in the era of complex, multi-million-gate designs, cube representation of a Boolean function becomes more and more impractical. In the following sections, we shall discuss some other, more efficient forms to represent Boolean functions.

A.2 Functional Expansion and Decision Diagrams

A canonical representation of a Boolean function can be obtained through various functional expansions.

Definition 1 (Shannon expansion) [2] *A Boolean function $f(x_1, x_2, \cdots, x_i, \cdots, x_n)$ can be expressed as*

$$f(x_1, x_2, \cdots, x_i, \cdots, x_n) \quad = \quad x_i f_i^1 + x_i' f_i^0;$$

where f_i^1 and f_i^0 are referred to as the positive and negative cofactor of f w.r.t. variable x_i.

Shannon expansion provides the most fundamental way to decompose a Boolean function. Many other, more

[2]In 1854 Boole [6] first described this type of expansion that was later incorrectly credited to Shannon.

general decomposition methods can be derived from it. For example, it can be verified easily that the traditional *minterm* representation can be obtained through iterative Shannon expansion. Shannon expansion also provides an important theoretical foundation for Binary Decision Diagrams (BDDs) [7], [8], which are the main theme of this paper. BDD's will be introduced in Section II-B.

Definition 2 (Orthonormal Expansion) *A Boolean function can be expanded using an orthonormal basis [9]. Let* ϕ_i, $i = \{1, 2, \ldots, k\}$, *be a set of Boolean functions such that* $\sum_{i=1}^{k} \phi_i = 1$, *and* $\phi_i \cdot \phi_j = \emptyset$, $\forall i \neq j \in \{1, 2, \ldots, k\}$. *Then any Boolean function* f *can be expanded as,*

$$f = \sum_{i=1}^{k} \phi_i \cdot f_{\phi_i}$$

The term f_{ϕ_i} *is called the generalized cofactor of* f *w.r.t.* ϕ_i.

Note that the Shannon expansion is a special case of *orthonormal* expansion when ϕ_i is a single variable. It will become clear in the following chapters that the decomposition based on Shannon expansion is limited to a single BDD node, while the decomposition based on an orthonormal expansion is based on a group of BDD nodes. The latter one, in most cases, will produce more efficient decompositions. This issue will be discussed in detail in Section IV-F.

A.3 Functional Decomposition

The purpose of *functional decomposition* is to break a large Boolean function into smaller parts, each of which can be implemented by a Boolean logic of manageable complexity. While, according to Kohavi [10], "functional decomposition is an *intrinsic* property of switching functions", finding a good decomposition is not trivial. Functional decomposition has been one of the most active research topics for decades. The problem of functional decomposition, as defined by Ashenhurst [11], Roth and Karp [12], can be formulated as follows.

Definition 3: *The goal of functional decomposition is to find a Boolean function,* $\mathbf{\Phi}(Y)$, *such that a Boolean function* $f(X)$ *can be expressed as*

$$f(X) = F(\mathbf{\Phi}_0(Y), \mathbf{\Phi}_1(Y), \cdots, \mathbf{\Phi}_k(Y), Z)$$

where $X = \{x_0, x_1, \cdots, x_n\}; Y = \{y_0, y_1, \cdots, y_s\}; Z = \{z_0, z_1, \cdots, z_m\}; Y, Z \subset X$. *If* $Y \cap Z = \emptyset$, *the decomposition is called disjunctive; otherwise it is conjunctive.* Y *is referred to as bound set, and* Z *is referred to as free set.*

Usually the decomposition can be dramatically simplified if a *disjunctive* decomposition can be found. Therefore, *disjunctive* decomposition has been the target of intensive research.

The first systematic approach to find disjunctive decomposition was proposed by Ashenhurst [11]. In his method, all variables are first partitioned into a *bound set* (Y) and a *free set* (Z). The Boolean function is then represented as a *Boolean matrix* (also called a *decomposition chart*) by using the variables in the bound set and the free set as column and row indices respectively. A disjunctive decomposition, $f(X) = F(\mathbf{\Phi}(Y), Z)$, where $Y \cap Z = \emptyset$, exists if the number of distinct columns (called column multiplicity) $\mu = 2$. A decomposition chart for function $F = w'x'z' + wx'z + w'yz + wyz'$ is shown in Fig. 2(a), with the bound set $Y = \{w, z\}$, and the free set $Z = \{x, y\}$. It can be found easily that the number of distinct columns is 2. As a result, F can be disjunctively decomposed as $F = f(\mathbf{\Phi}(w, z), x, y) = \mathbf{\Phi}x' + \mathbf{\Phi}'y$, where $\mathbf{\Phi} = wz + w'z'$. However, if the variables are partitioned such that the bound set $Y = \{y, z\}$, and the free set $Z = \{x, w\}$, the disjunctive decomposition will not be found. The decomposition chart corresponding to the latter case is shown in Fig. 2(b).

xy \ wz	0	1	2	3
0	1	0	0	1
1	1	1	1	1
2	0	0	0	0
3	0	1	1	0

(a) $\mu = 2$

xw \ yz	0	1	2	3
0	1	0	1	1
1	0	1	1	1
2	0	0	0	1
3	0	0	1	0

(b) $\mu > 2$

Fig. 2. Two decomposition charts for function $F = w'x'z' + wx'z + w'yz + wyz'$.

Ashenhurst's approach was extended by several other researchers [13], [14]. All these approaches are hence classified as *Ashenhurst/Curtis* decomposition methods. The common characteristics of these methods is that they all depend on Boolean matrix representation or on decomposition charts [14]. The drawback of these approaches is obvious. For an n-input Boolean function, there are $O(2^n)$ matrices or charts. Therefore, these methods are not practical from the engineering perspective.

Roth and Karp [12] proposed the first practical functional decomposition method which was later referred to as the *Roth-Karp* decomposition. In their method, a Boolean function is represented as a set of *cubes*. Compared with the representation based on Boolean matrix, cube representation is much more compact and has a capability to represent larger Boolean functions. The technique used in *Roth-Karp* decomposition is based on the partitioning of cubes into *compatible classes*. It should be noted that there is a one to one relation between the number of distinct columns in a decomposition chart and a compatible class. Therefore, there is no fundamental difference between *Ashenhurst* and *Roth-Karp* decomposition. However, due to a more efficient way to represent Boolean functions, *Roth-Karp* decomposition is much more efficient than *Ashenhurst* decomposition.

In view of the development of functional decomposition, it is interesting to note that the efficiency of a decomposition approach is well correlated with the representation of Boolean functions. This simple observation can be extended to note that a more efficient functional decomposition method, may become available when a Boolean representation form, more compact than *cube* representation, can be found.

In the following section, the most efficient representation of a Boolean function to date, a Binary Decision Diagram (BDD), is introduced.

B. Binary Decision Diagrams

The concept of binary decision diagrams (BDDs) was first proposed by Lee [7] in 1959. Lee demonstrated that a switching function can be efficiently implemented as a series of local decisions ($\mathbf{T} \quad x; A, B$), where A is "taken" if variable x is 0; and B is "taken" if x is 1. Lee pointed out the advantage of binary decision programs over an algebraic representation. He also pointed out that binary decision programs can be used for circuit synthesis [3].

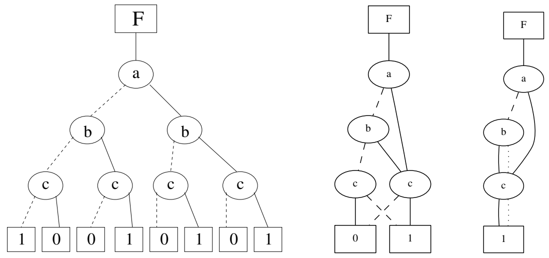
In 1978, Akers [8] first adopted the term "binary decision diagram". He also presented the first set of rules to reduce a BDD. However, BDDs had not been widely acknowledged until a set of efficient operators were proposed by Bryant [15] in 1986. Since then, the research and development of BDDs have achieved tremendous advance. Thousands of technical papers, research projects, and BDD packages contribute to the understanding and efficient manipulation of BDDs. BDDs have been applied to almost every aspect of VLSI CAD. They proved to be the most efficient Boolean representation to date.

B.1 Construction and Reduction of a BDD

In the fundamental work of Lee and Akers, no explicit assumption has been made about the variable reordering. Bryant [15] showed that under a fixed variable order, efficient algorithms can be devised to manipulate BDDs. A BDD under this restriction is generally referred to as an *ordered* BDD (OBDD). The OBDD for a Boolean function can be constructed using iterative Shannon expansion. For example, Fig. 3(a) shows the OBDD of a Boolean function, $F = ac + bc + a'b'c'$, with the variable order a, b, c. Each node of the OBDD corresponds to a Shannon expansion w.r.t. a single variable. The positive co-factor computed at a given node is generally represented by a *1-edge* (solid), while the negative co-factor is represented by a *0-edge* (dashed).

An OBDD is said to be *reduced* OBDD (ROBDD) if the following two reduction rules have been applied: **1**) node v is removed if its *1-edge* and *0-edge* point to the same node;

[3]Quoted from [7]: "It has been amply clear that, although Boolean representation of switching circuits has been the foundation on which switching theory had been built, the inherent limitations in the Boolean language seem to be difficult hurdles to surmount. Boolean representation is algebraic and highly systematic, but so inflexible that it is powerless against all but series-parallel circuits. [...] Binary-decision programming is our attempt of a way to get beyond these limitations. It works well for computation. Further studies will be required to find efficient ways of minimizing binary-decision programs and to make binary-decision programming an instrument for circuit synthesis."



(a) (b) (c)

Fig. 3. BDD Reduction Rules. (a) The OBDD obtained through Shannon expansion. (b) ROBDD. (c) ROBDD with *complement* edges.

2) a *subgraph* is removed if it is isomorphic to another subgraph. The ROBDD for function F in Fig. 3(a) is shown in Fig. 3(b). It should be noted that these two reduction rules are implicitly related to Boolean operations. Rule 1 corresponds to Boolean simplification, $af + a'f = f$; Rule 2 corresponds to a simple factorization, $af + bf = (a + b)f$. Therefore, the OBDD reduction provides a natural means for implicit Boolean simplification and factorization. As a result, a ROBDD is an implicitly factored Boolean representation. Bryant [15] proved that the Boolean representation based on ROBDD is *canonical*. In the rest of this thesis, ROBDD is referred to as a BDD for short.

In addition to the above reduction rules, the size of a BDD can be further reduced using a concept of *complement edges*. This concept was first introduced by Akers [8], and was efficiently implemented by Brace, Rudell and Bryant [16]. Basically, a complement edge points to the complementary form of a function (BDD node).

B.2 Variable Reordering

It is known that the size of a BDD is very sensitive to the variable order. A random, or carelessly chosen variable order will frequently result in an exponential size of the BDD. A common procedure to construct a BDD is as follows. First, an initial variable order is determined and the BDD is constructed according to that order. Then, a variable reordering algorithm is invoked to further minimize the size of BDDs.

Several heuristics have been proposed to provide an initial variable order. They mainly depend on the topological and variable dependence analysis in a Boolean network [17], [18]. Although these heuristics achieved a significant improvement over random ordering, the size of a BDD can be further reduced through variable *reordering*. Many heuristic variable reordering algorithms have been proposed. Most of these algorithms depend on a fundamental operation, *adjacent variable swapping* [19], [20], [21]. The most efficient algorithm, *sifting*, was proposed by Rudell [21]. He also proposed a mechanism called *dynamic variable reordering*, which allows a BDD to be reordered during the process of its construction. This approach partially relieves the so-called *memory blow-up* problem, which is caused by large intermediate BDDs.

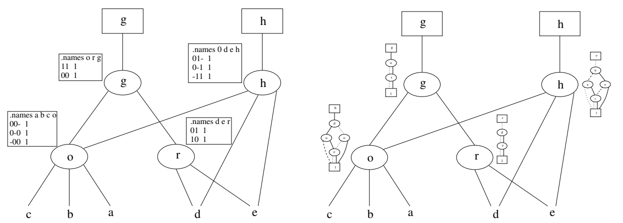
B.3 Don't Care Minimization

The problem addressed by BDD *don't care* minimization can be stated as follows. Given two completely specified Boolean functions f and c, with the *off-set* of c being *don't care* to f, find a Boolean function, denoted f/c, defined in the range $[f \cdot c, \quad f + c']$, such that the size of BDD of f/c is minimum.

This problem has been proved to be NP-complete [22]. Among all the proposed heuristics to perform such operation [23], [22], [24], [25], *RESTRICT* operator proposed by Coudert [23], is the most efficient one.

C. Boolean Network

A Boolean network is a *directed acyclic graph* (DAG); the representation of its structure is straightforward. Various Boolean network presentations differ mainly in the way they represent the *local function* pertaining to each *Boolean node*. Fig. 4(a) shows the Boolean network representation of SIS [2], in which the functionality of each Boolean node is represented as a set of SOP terms. This representation is commonly referred to as *multi-level SOP* representation.



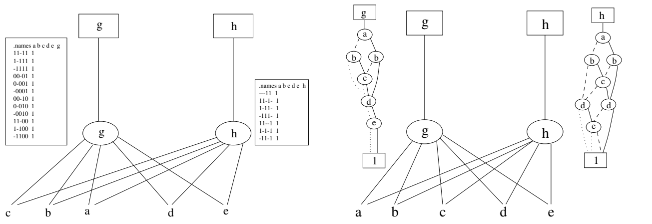
(a) Multi-level SOP representation (b) Local BDD representation

Fig. 4. The cube and BDD representations of Boolean nodes in the Boolean network.

The functionality of a Boolean node can also be represented as a BDD, as shown in Fig 4(b). This representation is known as a *local BDD* representation. Compared with the multi-level SOP representation, local BDD representation is relatively free of redundancy, because the redundancy inherent in the SOP form has been removed during the process of BDD construction. It also allows for a possible sharing between different Boolean nodes. Therefore BDD representation may potentially consume less memory than SOP.

A Boolean network can be also represented in a global form. In a global representation, Boolean network is collapsed into a set of global nodes, one node per primary output. Each global node depends only on primary inputs.

Fig. 5 shows two different global representations. In Fig. 5(a), each global node is represented in a *two-level* from. In Fig. 5(b), each global node is represented as a *monolithic* BDD. We refer to this representation as a *global* BDD representation. The advantage of BDD form becomes now obvious. Usually the logic redundancy embedded in a multi-level configuration can be completely removed by collapsing the Boolean network into two-level SOP or global BDD forms. However, such representation is not amenable to large Boolean networks, in which the size of representation may blow up. This issue will be further discussed in Chapter VI.



(a) Two-level SOP representation (b) Global BDD representation

Fig. 5. Two-level cube and monolithic BDD representation of a Boolean network.

III. Terminology

To facilitate the discussion in the sequel, we need to define a fundamental terminology and develop basic theorems related to the different operations on a BDD.

Definition 4 (BDD) *A BDD is a directed acyclic graph (DAG) representing a Boolean function. It can be uniquely defined as a tuple,* $\mathbf{BDD} = (\mathbf{\Phi}, \mathbf{V}, \mathbf{E}, \{\mathbf{0}, \mathbf{1}\})$, *where* $\mathbf{\Phi}$ *is the function node (root),* $\mathbf{V}$ *is the set of internal nodes,* $\mathbf{E}$ *is a set of edges, and* $\mathbf{0}$, $\mathbf{1}$ *are the terminal nodes.* □

Definition 5 (Leaf edges) *The leaf edge is an edge* $e \in \mathbf{E}$ *which is directly connected to a terminal node of the BDD. The set of leaf edges, denoted* $\mathbf{\Sigma}$, *can be partitioned into* $\mathbf{\Sigma_0}$, *the set of leaf edges connected to* $\mathbf{0}$, *and* $\mathbf{\Sigma_1}$, *the set of leaf edges connected to* $\mathbf{1}$. □

Definition 6 (Paths) $\mathbf{\Pi_0}$ *is the set of all paths from the root to terminal node* $\mathbf{0}$. $\mathbf{\Pi_1}$ *is the set of all paths from the root to terminal node* $\mathbf{1}$. $\mathbf{\Pi} = \mathbf{\Pi_0} \bigcup \mathbf{\Pi_1}$ *is the set of all paths.* □

An obvious, but important property of a **BDD** is that its set $\mathbf{\Pi_1}$ ($\mathbf{\Pi_0}$) defines the on-set, X^{on} (off-set, X^{off}) of function f. Specifically, each path $p \in \mathbf{\Pi_1}$ ($p \in \mathbf{\Pi_0}$) represents a disjoint cube in the on-set X^{on} (off-set X^{off}) of f.

Theorem 1 (Internal Edge Property) *Every internal edge* $e \in (\mathbf{E} - \mathbf{\Sigma})$ *belongs to at least one path* $p_1 \in \mathbf{\Pi_1}$ *and one path* $p_0 \in \mathbf{\Pi_0}$. □

Proof: The theorem is proved by contradiction. Since BDD is a connected graph, every edge must belong to either $\mathbf{\Pi_0}$ or $\mathbf{\Pi_1}$. For an edge $e \in (\mathbf{E} - \mathbf{\Sigma})$, if every path p passing through e belongs to $\mathbf{\Pi_1}$, then all the nodes below e can be collapsed into $\mathbf{1}$, so that $e \in \mathbf{\Sigma_1}$. Hence the contradiction. Same reasoning applies to $\mathbf{\Pi_0}$.

Definition 7 (Cut) *A cut* $(\mathbf{D}, \mathbf{V} - \mathbf{D})$ *of a* **BDD** *is a partition of its nodes* $\mathbf{V}$ *into disjoint subsets* $\mathbf{D}$ *and* $(\mathbf{V} - \mathbf{D})$

such that root $\in$ **D** *and terminals* **0**, **1** $\subseteq$ (**V-D**). *A cut cannot cross any path* $p \in \mathbf{\Pi}$ *more than once. A horizontal cut is a cut in which the support*[4] *of* **D** *and* (**V** - **D**) *are disjoint.* □

Fig. 6 shows a BDD with several possible *cuts*. As described in the next chapter, horizontal cuts will be useful in performing the BDD decomposition.

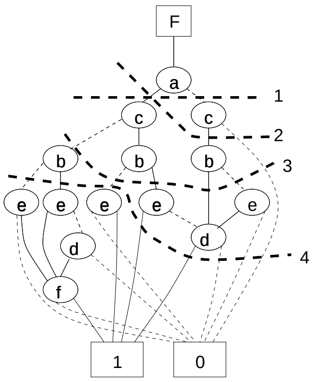


Fig. 6. Valid cuts on a BDD.

IV. Theory of BDD Decomposition

BDDs have drawn a lot of attention from the logic synthesis and verification research community. This can be attributed to their excellent capability for the concise representation and efficient manipulation of Boolean functions. However, most known BDD decomposition methods employ BDD as an efficient platform to carry out traditional decompositions, such as *Ashenhurst* [11] and *Roth-Karp* [12] decompositions, and do not utilize the full capability of BDDs.

BDDs are fundamentally different from traditional cube forms. In a cube form, a Boolean function is represented as a set of *individual* cubes. The relationship between different cubes is not clear until certain rules of Boolean algebra are applied. For example, the fact that exists a literal c common to two cubes $\{ac\}$ and $\{bc\}$ is not aaparent until some sort of factorization is applied. In contrast to that, BDDs have a *collective* power to represent Boolean functions, and the relationship between different *paths* in a BDD (*i.e.* cubes) is obvious. Therefore, instead of performing traditional functional decomposition using BDDs solely as a platform, decomposition methods specifically tailored for BDDs should be developed. Since BDD is a *directed acylic graph*, in order to uncover the decomposition encoded in such collectively represented Boolean function, some kind of graph traversal or structural analysis techniques are necessary.

In this chapter, a BDD decomposition theory which is based on BDD structural analysis is presented.

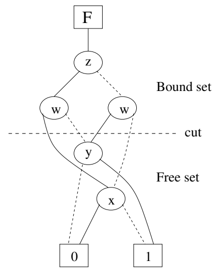
A. Previous Work

The majority of current BDD decomposition methods relies on two important properties of BDDs: 1) BDD is used as an efficient representation of a Boolean function; 2) The structure of a BDD is implicitly related to the decomposition chart used by *Ashenhurst* decomposition [11]; specifically, the partitioning of variables into a *bound set* and a *free set* is directly related to the variable ordering in the BDD. The following example illustrates this idea.

Example 1: *Consider function* $F = w'x'z' + wx'z + w'yz + wyz'$. *The decomposition chart (Fig. 2(a)), leading to a disjunctive functional decomposition of this function, is re-drawn in Fig. 7(a). For the purpose of comparison, the reordered BDD for function F is shown in Fig. 7(b). A cut in the BDD partitions the variables into a bound set and a free set. Notice that the variable partitioning is exactly the same as that in Fig. 7(a), with the bound set* $\{z, w\}$ *and free set* $\{y, x\}$. *This means that a good variable partitioning for disjunctive Ashenhurst decomposition can also be obtained implicitly through a BDD variable reordering.* □

xy \ wz	0	1	2	3
0	1	0	0	1
1	1	1	1	1
2	0	0	0	0
3	0	1	1	0

(a)



(b)

Fig. 7. Decomposition chart and BDD of a Boolean function.

Consider a cut in a BDD, which partitions a set of BDD nodes V into two sets, D and $(\mathbf{V} - \mathbf{D})$. A node $v \in (\mathbf{V} - \mathbf{D})$ which is connected to an edge in the cut in called a *cut-node*. A set of cut-nodes associated with a given cut is called *cut-node-set*. In Fig. 7(b), *cut-node-set* $= \{y, x\}$. An important observation is that the cardinality of the *cut-node-set* determines the total number of distinct columns in the decomposition chart. This can be explained as follows. In Fig. 7(b), any path from the *root* to terminals must go through either y or x. Therefore, if z, w are treated as column indices and y, x as row indices, the number of distinct columns is exactly two.

The decomposition process begins by encoding the BDD nodes in the *cut-node-set*. This is shown in Fig. 8. The number of bits (variables) required for the encoding is $log_2(n)$, where n is the cardinality of the *cut-node-set*. For this example, one bit (variable) is sufficient. A new variable, g, is introduced. The BDD of g can be obtained by substituting y and x with their respective codes, as shown in Fig 8(a). This results in the final Ashenhurst decomposition, $F = gx + g'y$, where $g = zw + z'w'$.

Although an optimal decomposition for the above function can be found by the methods, it is not the case for general, complex Boolean functions. Due to lack of a criterion for a *good* cut, a cut is usually performed when the number of variables above the cut is less

[4] *support* is defined as the set of variables a Boolean function depends.

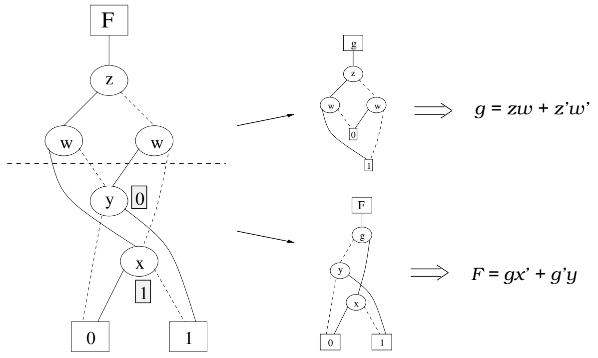


Fig. 8. Ashenhurst decomposition using BDD.

than some fixed value, k. The application of these methods are hence restricted to Look-Up-Table (LUT)-based FPGAs, with k being the number of inputs to an FPGA block [26], [27]. We believe that, with the help of structural analysis of BDDs, this type of decomposition can be extended to decompositions leading to efficient multi-level implementations.

We are also aware of an approach in which a BDD is used as an indirect form to uncover good decompositions. In [28], a subset of spectral coefficients of a Boolean function, represented as a BDD, is calculated. The BDD is then decomposed through the examination of certain properties of the subset. Since the calculation of spectral coefficients is very expensive, this method is potentially computationally intensive.

Finding an efficient multi-level representation of a Boolean function by analyzing the structure of its BDD was first studied by Karplus [29] at the early ages of BDDs. He introduced the concept of a *1-* and *0-dominator*[5], which lead to an algebraic AND/OR decomposition. Fig. 9 illustrates the concept of a *1-* and a *0-dominator*. Basically, a 1-dominator (0-dominator) is a node which belongs to every path $p \in \mathbf{\Pi_1}$ ($p \in \mathbf{\Pi_0}$). The existence of 1-dominator (0-dominator) allows the BDD to be decomposed into two parts conjunctively (disjunctively).

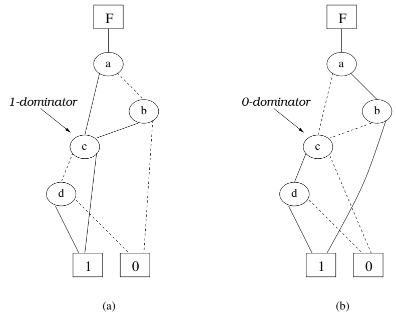


Fig. 9. Example of *1-* and *0-dominators* introduced by Karplus. (a) 1-dominator leads to an algebraic conjunctive decomposition, $F = (a + b)(c + d)$. (b) 0-dominator leads to an algebraic disjunctive decomposition, $F = ab + cd$.

[5]Both *1-* and *0-dominators* are special cases of our *generalized dominator*, discussed in Section IV-C

Since Karplus [29], very little work has been reported in this area. As far as we know, there have been at least two attempts to perform logic optimization targeting multi-level representations by analyzing BDD structures. Bertacco *et al.* [30] proposed a method which performs hierarchical disjunctive decomposition directly on a BDD. This method basically annotates disjunctive decomposition inherent in the BDD structure. Compared with SIS, their method is faster and generates much better results on some circuits. However, their method fails to generate good decompositions on BDDs with complement edges. Stanion *et al.* [31] proposed a *generalized cofactor*-based Boolean division and factorization method. Given a divisor D, a function F can be written as $F = D \cdot cof(F, D) + D' \cdot cof(F, D')$. Consequently, Boolean division is performed by setting $Q = cof(F, D)$ and $R = D' \cdot cof(F, D')$. The result can be further improved by realizing that Q, D and R imply *don't care* sets to each other. However, due to a lack of efficient way to generate Boolean divisors, the improvement of this method over SIS is marginal. Neither of the above-mentioned methods address a general decomposition of BDDs onto expressions involving XOR logic.

B. Fundamentals

Before diving into the details of different types of BDD decompositions, let us first provide a theoretical analysis of two fundamental decompositions, namely Boolean division and Boolean subtraction. All other types of decompositions can be derived from these two.

Definition 8 (Boolean Division) *Function D is a Boolean divisor of F if there exists a function Q, called quotient, such that $F = QD$.*

In [32], a Boolean division is defined as $F = QD + R$, and D is called a *Boolean factor*. In our decomposition scheme, we always assume $R = \emptyset$. To comply with the terms "division", we call D a *Boolean divisor*, instead of a *factor*. In this paper, we shall use the terms *Boolean division*, *conjunctive Boolean decomposition*, and *Boolean AND decomposition*, interchangeably.

Definition 9 (Boolean Subtraction) *Function D is a Boolean subtractor of F if there exists a function R, called remainder, such that $F = D + R$.*

In the sequel, we shall use the terms *Boolean subtraction*, *disjunctive Boolean decomposition*, and *Boolean OR decomposition*, interchangeably.

Theorem 2: [33] *Function D is a Boolean divisor of F if and only if $F \subseteq D$.*

Proof: *If D is a Boolean divisor of F, then there exists Q such that $F = QD$; $QD \neq \emptyset \Rightarrow F \subseteq D$. On the other hand, $F \subseteq D \Rightarrow F = DF = D(F + R) = DQ$. Here R is any function $R \subseteq D'$.*

Example 2: Consider two functions $F = e + bd$ and $D = e + d$. Since the on-set of F is covered by that of D, *i.e.* $F \subset D$, D is a Boolean divisor of F. Function F can be decomposed as $F = (e + d)(e + b)$. The BDDs for function F and D are shown in Fig. 11. □

Theorem 3: *A function D is a Boolean subtractor of F if and only if $F \supseteq D$.*

Proof: *The proof is dual of that of Theorem 2.*

Definition 10 (Co-factoring sequence) *Consider a node v in a BDD. The path from root to v can be uniquely defined as a set of variables, where each variable may appear in true or complemented form. Such a set of variables is called a co-factoring sequence. If v is a terminal node, the list is called a terminal co-factoring sequence.*

Let us now study the properties of Boolean functions D and Q in order to satisfy $F = QD$. Since BDD is a graphical representation of a sequence of Shannon expansions of the Boolean function, the process can be readily demonstrated by using a sequence of co-factoring operations. F can be expanded using Shannon expansion as:

$$F = x'F_{x'} + xF_x \quad (1)$$

QD can be expanded in the same way, we obtain

$$QD = (x'Q_{x'} + xQ_x)(x'D_{x'} + xD_x) = x'Q_{x'}D_{x'} + xQ_xD_x \quad (2)$$

Then $F = QD$ if the following two conditions are satisfied,

$$\begin{aligned} Q_{x'}D_{x'} &= F_{x'} \\ Q_xD_x &= F_x \end{aligned} \quad (3)$$

By induction, the above conditions can be generalized to any co-factoring sequence w. That is, if

$$Q_wD_w = F_w \quad (4)$$

is true for any co-factoring sequence w, then $F = QD$.

When F, Q and D are represented as BDDs, to check whether $F = QD$ is true, only terminals need to be checked to see if condition $Q_wD_w = F_w$ is satisfied. When w is a terminal co-factoring sequence, for condition $F_w = Q_wD_w$ to be true, D_w and Q_w must satisfy the following two conditions,

$$F_w = 1 \;\Rightarrow\; D_w = 1, \quad Q_w = 1. \quad (5)$$

$$F_w = 0 \;\Rightarrow\; D_w = 0, \quad Q_w = * \quad (D_w = *, Q_w = 0). \quad (6)$$

where * stands for don't care.

Theorem 4 (Boolean divisor condition) *D is a conjunctive Boolean divisor of F, if for every terminal co-factoring sequence w for a given variable ordering,*

$$\begin{aligned} F_w = 1 &\;\Rightarrow\; D_w = 1 \\ D_w = 0 &\;\Rightarrow\; F_w = 0 \end{aligned} \quad (7)$$

Proof: *Since $F_w = 1 \Rightarrow D_w = 1$ for any terminal co-factoring sequence w, $X_D^{on} \supseteq X_F^{on}$. Hence D is a Boolean divisor of F.* □

Theorem 4 provides an efficient way to check whether a Boolean function D is a divisor of another Boolean function F. As will be become clear in the following sections, Theorem 4 provides the theoretical foundation for *generalized dominator*.

In the same manner, the condition for a Boolean subtractor can also be formulated.

Theorem 5 (Boolean subtractor) *D is a disjunctive Boolean subtractor of* $\mathbf{F}$, *if every terminal co-factoring sequence w for a given variable ordering,*

$$\begin{aligned} D_w = 1 &\;\Rightarrow\; F_w = 1 \\ F_w = 0 &\;\Rightarrow\; D_w = 0 \end{aligned} \quad (8)$$

Proof: *The proof is dual of that of Theorem 4.* □

C. AND/OR Decomposition

In this section, different types of BDD decompositions targeting AND/OR logic decomposition are presented.

C.1 Boolean Decomposition

First, the most general structure leading to a Boolean AND/OR decomposition is examined. This structure is referred to as a *generalized dominator*.

Definition 11 (Generalized Dominator) *Consider a cut partitioning the set of BDD nodes of function F into D and (V-D). The portion of the BDD defined by D is copied to form a separate graph. In that graph, an edge e is connected to* **0** *if $e \in \mathbf{\Sigma_0}$ in the original BDD of F, and it is connected to* **1** *if $e \in \mathbf{\Sigma_1}$ in the original BDD of F. All the internal edges $e \in (\mathbf{E} - \mathbf{\Sigma})$ are left dangling. The resulting graph is called a generalized dominator* $\mathbf{\Lambda}$. □

Fig 10 shows the construction of a *generalized dominator*. In Fig. 10(a), a cut is performed on the BDD. Then the portion above the cut is copied to a separate graph, which is shown in Fig. 10(b). The construction is completed by connecting $\mathbf{\Sigma}$ edges of the graph to the corresponding terminals in the original BDD. Note that because of the dangling edges, a *generalized dominator* $\mathbf{\Lambda}$ is not a BDD. By assigning the dangling edges to different constant value (1 or 0), $\mathbf{\Lambda}$ can be used to decompose a BDD conjunctively or disjunctively. Let η be a set of all dangling edges.

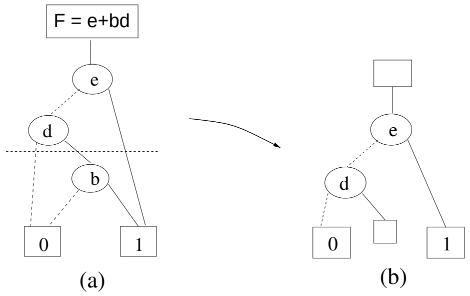


Fig. 10. Generation of a Boolean divisor based on *generalized dominator*.

The following theorem shows how to obtain a Boolean divisor and perform the division[6] by redirecting the dangling edges η of a $\mathbf{\Lambda}$ to constant node **1**.

Theorem 6 (Construction of Q, D) *Given a generalized dominator* $\mathbf{\Lambda}$ *of function* $\mathbf{F}$, *the Boolean divisor* $\mathbf{D}$ *is obtained*

[6] We also refer to it as a Boolean AND decomposition

from $\mathbf{\Lambda}$ *by redirecting dangling edges* $e \in \eta$ *of* $\mathbf{\Lambda}$ *to* $\mathbf{1}$. *The quotient* Q *is obtained by minimizing* F *with the off-set of* $\mathbf{D}$ *as a don't care set.*

Proof: *According to Theorem 1, there is at least one path* $p \in \mathbf{\Pi_1}$ *passing through each internal edge of* $\mathbf{\Lambda}$. *By redirecting these internal edges of* $\mathbf{\Lambda}$ *to* $\mathbf{1}$, *the obtained BDD (function)* D *covers all paths* $p \in \mathbf{\Pi_1}$, *i.e.* $X_D^{on} \supset X_F^{on}$. *Therefore,* D *is a Boolean divisor for function* F *(see Theorem 2).*

The quotient Q *can be verified by checking whether the condition* $F_w = Q_w D_w$ *is true for all possible paths* w *in the BDD. Recall that* Q *is a copy of* F, *except for the edges which correspond to the off-set of* D. *Therefore, the on-set of non-minimized* Q, $X_Q^{on} = X_F^{on}$. *Since* $X_D^{on} \supset X_F^{on}$, $Q_w D_w = F_w$ *whenever* $F_w = 1$. *By construction,* $D_w = 0 \Rightarrow F_w = 0$, *and* Q_w *is set to don't care, so that* $X_Q^{on} \supseteq X_F^{on}$. *Therefore,* $Q_w D_w = F_w$ *when* $F_w = 0$. Q *is the quotient of this Boolean division.* □

Example 3: *To illustrate the Theorem 6, a simple example is shown in Fig. 11. The BDD of function* $F = e + bd$ *is shown in Fig. 11(a). First, a cut is performed on the BDD. Then, a generalized dominator is constructed based on the cut. Since a Boolean division is anticipated, the dangling edges on the generalized dominator are redirected to constant* $\mathbf{1}$. *The Boolean divisor* D *is easily evaluated as* $D = e + d$. *The quotient* Q *of this division can be obtained from* F *by setting the off-set* $\{e'd'\}$ *of* D *as a don't care. After minimization of* F *with this don't care,* $Q = e + b$. *Notice that* $(D = e + d) \supset (F = e + bd)$, *and* $(Q = e + b) \supset (F = e + bd)$.

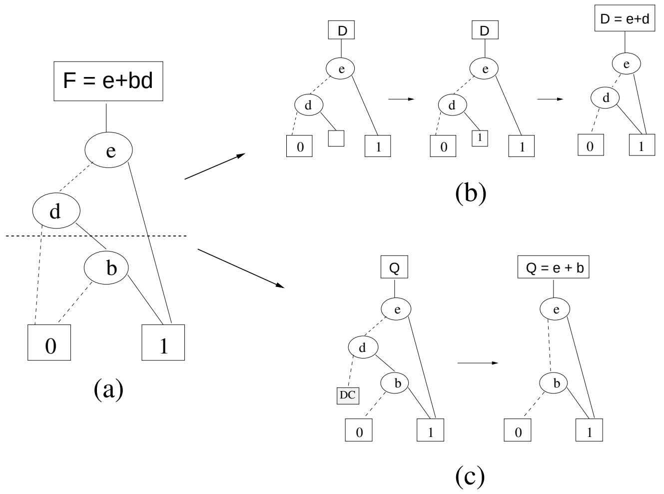


Fig. 11. A simple example of Boolean division.

In the following, a more complex example is provided.

Example 4: *A complete conjunctive (AND) decomposition, including the construction of a quotient* Q, *is shown in Fig. 12. In Fig. 12(a), a cut is performed in the BDD. In Fig. 12(b), the generalized-dominator is obtained by copying the portion above that cut to a graph. Then a Boolean divisor is built by redirecting all the dangling edges of that graph to* $\mathbf{1}$. *The reduced BDD of* D *is also shown in Fig. 12(b). As indicated in the figure, this decomposition exposes a 0-dominator in* D, *which was not present in the original BDD of* F. *Therefore,* D *can be easily decomposed as* $D = (af + b + c)$. *In Fig. 12(c), quotient* Q *is obtained from* F *by minimizing function* F *using function* D *as a* don't care. *This results in* $Q = (ag + d + e)$. *As a result of this process, the whole function can be decomposed as* $F = (af + b + c)(ag + d + e)$. □

Boolean subtraction is the dual case of Boolean division. The following is the fundamental theorem for Boolean subtraction.

Theorem 7 (Construction of D, R) *Given a generalized dominator* $\mathbf{\Lambda}$ *of function* F, *the Boolean subtractor* D *of* F *can be obtained by redirecting dangling edges* $e \in \eta$ *of* $\mathbf{\Lambda}$ *to* $\mathbf{0}$. *The remainder* R *is obtained by minimizing* F *using the on-set of* D *as a don't care set.*

Proof: *According to Theorem 1, there is at least one path* $p \in \mathbf{\Pi_0}$ *passing through each internal edge of* $\mathbf{\Lambda}$. *By redirecting these internal edges of* $\mathbf{\Lambda}$ *to* $\mathbf{0}$, *the BDD of the resulting function* D *covers all paths* $p \in \mathbf{\Pi_0}$, *i.e.* $X_D^{off} \supset X_F^{off}$ $(X_D^{on} \subset X_F^{on})$. *Therefore,* $\mathbf{D}$ *is a Boolean subtractor for function* F *(see Theorem 2). The rest of the proof is dual of that of Theorem 6.* □

During the process of finding an optimal Boolean AND/OR decomposition, all possible cuts should be exercised. Obviously, the number of possible cuts could be very large even for a medium size BDD. Therefore, some filtering mechanism to reduce the number of candidate cuts should be developed. In the following, several filters have been identified to disqualify cuts which are *invalid* or *redundant*.

Definition 12 (Valid cut) *A cut is called valid if it contains at least one edge* $e \in \mathbf{\Sigma}$. *Otherwise, a cut is invalid.* □

Theorem 8: *Only valid cuts lead to nontrivial Boolean decomposition.*

Proof: *Consider an invalid cut in the BDD. By definition, the generalized dominator generated from the invalid cut does not have any* $\mathbf{\Sigma}$ *edges. Hence all terminal edges are dangling. Since all dangling edges are redirected to* $\mathbf{1}$ $(\mathbf{0})$, *the Boolean divisor (Boolean subtractor)* $D = 1$ $(D = 0)$. *These cases are shown in Fig. 13(b) and (c). Now consider a valid cut. Since* $\Sigma_1(\Sigma_0) \neq \emptyset$, *some of the terminal edges of the generalized dominator are connected* 0 (1), *while others* (η) *are connected to* 1 (0). *Hence the resulting* D *and* Q (R) *is nontrivial, leading to nontrivial decomposition.* □

Definition 13 (0-Equivalent Cuts) *Two cuts are 0-equivalent if they contain the same subset of* $\mathbf{\Sigma_0}$ *edges.* □

Definition 14 (1-Equivalent Cuts) *Two cuts are 1-equivalent if they contain the same subset of* $\mathbf{\Sigma_1}$ *edges.* □

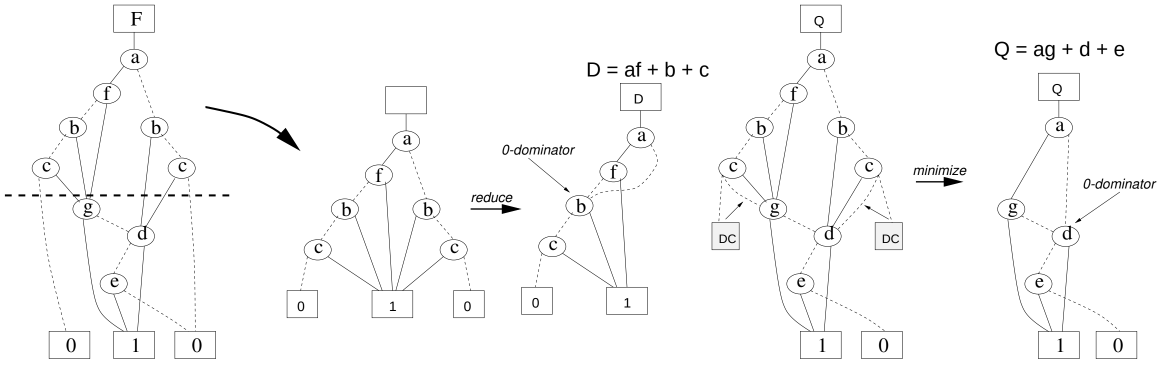
Theorem 9 (Distinct Cuts) *All Boolean divisors obtained from 0-equivalent cuts are identical.*

Proof: *Consider two cuts, a and b, which are 0-equivalent. In each of the Boolean divisors generated by those cuts, edges* $e \in \mathbf{\Sigma_0}$ *are connected to* $\mathbf{0}$. *All other edges are connected to* $\mathbf{1}$. *Hence, both Boolean divisors have the same paths from root to* $\mathbf{1}$ *(on-set) and the same paths from root to* $\mathbf{0}$ *(off-set). Hence, both Boolean divisors are identical.* □

This is illustrated in Fig. 13(d), (e), which showing that *cut* 2 and *cut* 3 belong to 0-equivalent class, and hence lead to identical Boolean divisors.

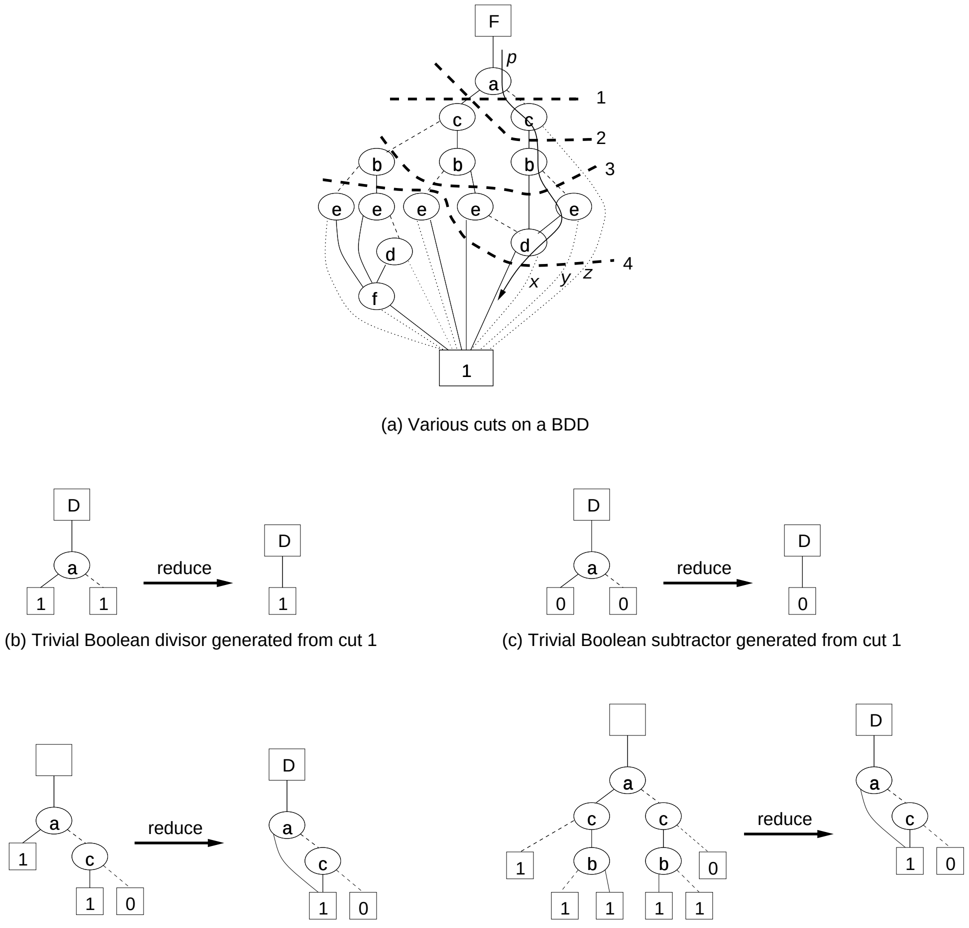
Theorem 10 (Distinct Cuts) *All Boolean subtractors obtained from 1-equivalent cuts are identical.*

Proof: *The proof is similar to that of Theorem 9.* □



(a) Original function F (b) Generalized dominator and Boolean divisor (c) Minimizing F with off-sets in D as don't care

Fig. 12. Obtaining a factored form on a BDD.



(a) Various cuts on a BDD

(b) Trivial Boolean divisor generated from cut 1 (c) Trivial Boolean subtractor generated from cut 1

(d) Boolean divisor generated from cut 2 (e) Boolean divisor generated from cut 3

Fig. 13. Effect of a cut on the generation of Boolean divisor/subtractor.

In conclusion, finding a cut can be viewed as a partitioning of $\mathbf{\Sigma_0}$ and $\mathbf{\Sigma_1}$ edges, rather than partitioning of BDD nodes. Therefore, the total number of all possible cuts is $2^{(|\mathbf{\Sigma_1}|+|\mathbf{\Sigma_0}|)}$. An in-depth analysis of BDD structure reveals that the actual number of valid cuts is much smaller. The number of valid cuts is further limited by the following cut property.

Theorem 11 (Transitive Cut Property) *Consider a node v, and its 1-edge (or 0-edge) $e \in \mathbf{\Sigma}$. A cut containing edge e must also contains all other $\mathbf{\Sigma}$ edges spanning[7] from a path from root to v.*

Proof: *The transitive property is guaranteed by the fact that a cut cannot cross the same path more than once. As shown in Fig. 13(a), three $\mathbf{\Sigma_0}$ edges, x, y and z originate (span) from nodes which are on the same path p to node v. Therefore, any cut crossing edge x must also cross edge y and z.* □

[7] An edge is said to span from a path if it is incident to a node on the path.

The *transitive cut property* dramatically decreases the number of possible cuts in a BDD. However, since the actual number of valid cuts depends on a specific BDD structure, it is difficult to a give a concrete formula for the total number of valid cuts.

In our approach we limit our attention to *horizontal cuts*. Our experience shows that *horizontal cuts* work well on most BDDs. Under the worst case, the total number of *horizontal cuts* is $|\mathbf{V}|$, where $\mathbf{V}$ is the number of variables (levels of a BDD). In practice, the total number of valid *horizontal cuts* is much smaller than $|\mathbf{V}|$, because many cuts are either 1-equivalent or 0-equivalent.

C.2 Algebraic Decomposition

Algebraic decomposition is a special case of Boolean decomposition. Due to the importance of the algebraic decomposition and easiness with which it can be identified on a BDD, algebraic AND/OR structures are readily identified independently of *generalized dominators*. Two basic structures leading to an algebraic AND/OR decomposition were found by Karplus [29]. Here we review these structures and show that they are special cases of our *generalized dominator*.

Definition 15 (1-Dominator) *Node $v \in \mathbf{V}$ which belongs to every path $p \in \mathbf{\Pi_1}$ is called a 1-dominator.* □

It should be noted that the above definition applies only to BDDs without complement edges above node v. A BDD with a *1-dominator* has been shown in Fig. 9(a).

Theorem 12 (Algebraic AND decomposition) *The BDD which contains a 1-dominator can be algebraically decomposed into two conjunctive parts, i.e., $F = f \cdot g$, where the supports of f and g are disjoint.*

Proof: *Fig. 14(a) shows the structure of a 1-dominator, in which node v lies on all path $\mathbf{\Pi_1}$. If a cut is be performed directly above node v, the Boolean divisor generated from the generalized dominator is structurally identical to the portion of the BDD above the cut. This is shown in Fig. 14(b). The quotient of this division can be obtained by redirecting the $\mathbf{\Sigma_0}$ edges to don't care, which can then be redirected to node v. Then all BDD nodes in part f have the same transitive child, v, and the whole f part collapses into node v. This is shown in Fig. 14(c). Since there is no common support between f and g, the decomposition is algebraic.* □

Definition 16 (0-Dominator) *Node $v \in \mathbf{V}$ which belongs to every path $p \in \mathbf{\Pi_0}$ is called a 0-dominator.* □

0-dominator is a dual of *1-dominator*. An example of a *0-dominator* is shown in Fig. 9(b).

Theorem 13 (Algebraic OR decomposition) *The BDD which contains a 0-dominator can be algebraically decomposed into two disjunctive parts, i.e., $F = f + g$, where the supports of f and g are disjoint.*

Proof: *The proof is similar to that of Theorem 12. It is illustrated in Fig. 15.* □

D. XOR Decomposition

BDD decomposition based on *generalized dominators*, described in the previous sections, relies on $\mathbf{\Sigma}$ edges. It is interesting to note certain properties of $\mathbf{\Sigma}$ edges. Namely, $\mathbf{\Sigma}$ edges provide an "early evaluation" of a Boolean function. For example, the value of function $f = ab$ ($f = a + b$) can be determined when either a or b equals to 0 (1). BDDs of functions that are mainly composed of AND/OR logic tend to have many $\mathbf{\Sigma}$ edges. On the other hand, BDDs of functions populated with XORs have very few or no $\mathbf{\Sigma}$ edges. Therefore, the value of a function with XORs is determined by the *relative* values of its variables. For example, the value of function $f = a \oplus b$ will only be determined when values of both variables a and b are given.

It is apparent that the decomposition which relies on $\mathbf{\Sigma}$ edges will fail on a BDD with few $\mathbf{\Sigma}$ edges. In this section, the techniques targeting XOR-type decomposition of a BDD are developed. In this case the *complement edges* are used to uncover the underlying XOR decomposition. The primary goal of introducing complement edges was to reduce the memory usage. Interestingly, we find that the presence of complement edges in a BDD is related to XOR decomposition. In the sequel, we will use XNOR ($\overline{\oplus}$) instead of XOR because XNOR has a more straightforward representation on BDDs.

D.1 Algebraic XNOR Decomposition

Definition 17 (x-dominator) *Node $v \in \mathbf{V}$ which is contained in every path $p \in \mathbf{\Pi}$ is called an x-dominator.* □

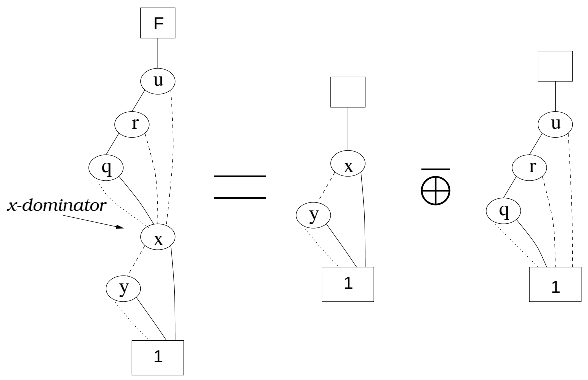
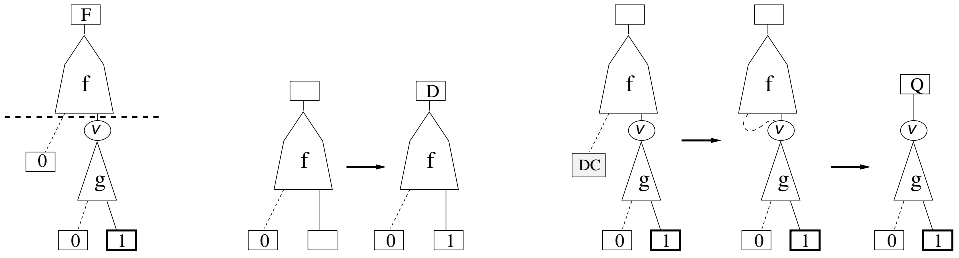


Fig. 16. Role of an *x-dominator* in XNOR decomposition.

A BDD with an *x-dominator* is shown in Fig. 16. Note that the definition of *x-dominator* implies that there must exists at least one *complement edge* above the *x-dominator* v. Otherwise all the BDD nodes above v will collapse into v. Therefore *x-dominators* do not exist on BDDs without *complement edges*.

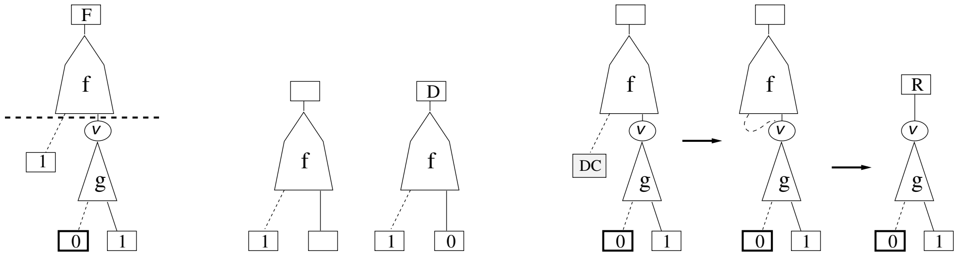
Theorem 14 (Algebraic XNOR decomposition) *Let v be an x-dominator of the BDD of function F. The BDD of F can be algebraically decomposed as $F = u \overline{\oplus} f$, where f is a BDD rooted at v, and u is the BDD rooted at the original function with v replaced by constant 1.*

Proof: *Fig. 17(a) shows a generic BDD with x-dominator v. By definition of complement edges, the BDD of f rooted at v can be split into two parts, f and f', as shown in Fig. 17(b). Then the BDD can be represented as a disjunction of two parts, as shown in Fig. 17(c). Note that f and f' are the 1-dominators in their respective BDDs. By defining u to be the BDD of F*



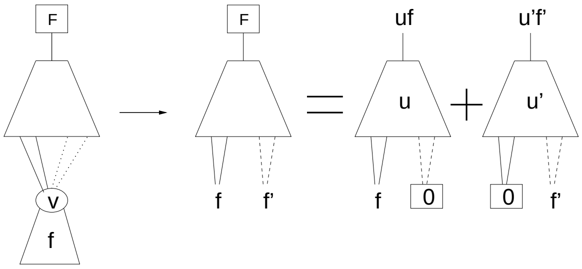
(a) 1-dominator structure (b) Generation of the divisor D (c) Generation of quotient Q

Fig. 14. 1-dominator structure and its corresponding decomposition.



(a) 0-dominator structure (b) Generation of the subtractor (c) Generation of remainder

Fig. 15. 0-dominator structure and its corresponding decomposition.



(a) (b) (c)

Fig. 17. x-dominator and its decomposition

in which v is replaced with 1, function F can be decomposed as $F = uf + u'f' = u\overline{\oplus}f$. □

Example 5: *An x-dominator is shown in Fig. 16. According to Theorem 14, the BDD can be algebraically decomposed as $F = (x + y)\overline{\oplus}(u' + r' + q)$.* □

D.2 Boolean XNOR Decomposition

The goal of Boolean XNOR decomposition of function F is to find a decomposition $F = f\overline{\oplus}g$ that will minimize the cost of its implementation. Usually XNOR decomposition is performed on a function in which good AND/OR decompositions are unlikely to be found.

Theorem 15 (Boolean XNOR decomposition) *For a Boolean function F, given an arbitrary Boolean function f, there always exists a Boolean function g, such that $F = f \overline{\oplus} g$.*

Proof: *The proof is trivial, using the following Boolean transformation.*

$$F = f \overline{\oplus} (f\overline{\oplus}F) = f \overline{\oplus} g \tag{9}$$

where f is an arbitrary Boolean function, and $g = f\overline{\oplus}F$. □

While exhaustive search for all possible functions f is clearly prohibitive, a set of good candidates for f can be detected directly form a BDD structure, called *generalized x-dominator*, defined as follows.

Definition 18 (Generalized x-dominator) *Node $v \in \mathbf{V}$ which is pointed to by both the complement and regular edges is called a generalized x-dominator. The complement edges associated with the generalized x-dominator are called XOR-related complement edges.* □

Let BDD of F contains a generalized x-dominator f. By performing transformation $g = f\overline{\oplus}F$, the *regular edges* pointing to f are redirected to 1 (because $f\overline{\oplus}f = 1$), and the *complement edges* pointing to f are redirected to 0 ($f\overline{\oplus}f' = 0$). In the process, the transformation removes the *XOR-related complement edges* pointing to f. The XNOR core of a Boolean function can be efficiently extracted by removing XOR-related complement edges from its BDD.

Example 6: *Figure 18 shows the BDD for circuit rnd4-1, a test case in the MCNC benchmark suite. According to Definition 18, there are two generalized x-dominators in this BDD, namely $f = x_1\overline{\oplus}x_4$, and $h = x_4$. We illustrate the decomposition based on f. Its BDD is shown in Figure 18(b). The BDD of $g = f\overline{\oplus}F$ is also shown in Figure 18(b). The BDD of f consists of an x-dominator, and the BDD of $g = f\overline{\oplus}F$ consists of 1- and 0-dominators. Therefore both of them can be further algebraically decomposed, resulting in $F = (x_1\overline{\oplus}x_4)\overline{\oplus}(x_2(x_3 + x_1x_4))$.* □

E. MUX Decomposition

BDD is a graphical representation of a sequence of Shannon expansions. Each node in a BDD can be viewed as a simple multiplexor (MUX). Taking MUX

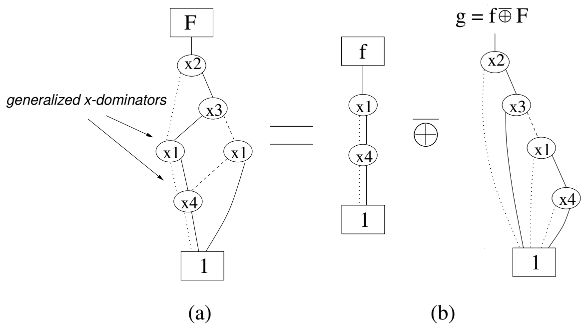


Fig. 18. XNOR decomposition of function *rnd4-1*

decomposition regardless of the specific BDD structure often leads to poor multi-level Boolean expressions. Simple MUX decomposition w.r.t a single node is only beneficial when the overlap between its two co-factors is less than a certain threshold. This case is shown in Fig. 19.

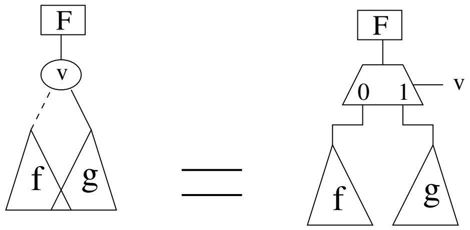


Fig. 19. Simple MUX decomposition

E.1 Functional MUX

The generalization of a simple MUX decomposition is referred to as *functional* MUX decomposition. In this decomposition, the control signal is a function, instead of a single variable. Functional MUX decomposition often leads to concise multi-level expressions.

Theorem 16 (Functional MUX Decomposition) *Consider a BDD structure, in which two nodes, u and v, cover all paths $p \in \mathbf{\Pi}$. The BDD can then be decomposed as $hg + h'f$, where h is obtained by redirecting node u to 1, and node v to 0, and f and g are functions associated with nodes u and v, respectively.*

Proof: *The proof is similar to that of Theorem 14. The decomposition is shown in Fig. 20.* □

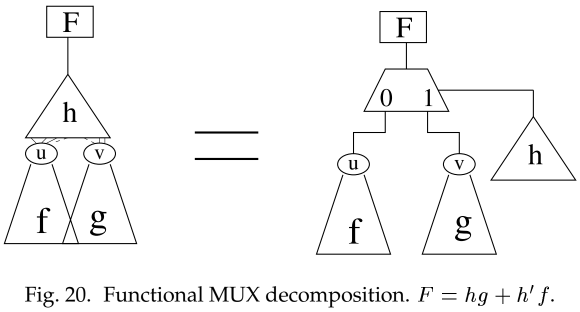


Fig. 20. Functional MUX decomposition. $F = hg + h'f$.

Similar to the definitions of the *0-* and *1-dominator*, this theorem applies only to BDDs without *complement edges* above u and v. While the *functional* MUX decomposition exists in various Boolean functions, they are especially common in arithmetic functions. They are frequently associated with XNOR decomposition.

Example 7: *Shown in Figure 21 is a simple example of a functional MUX decomposition. Nodes u and v cover all paths $p \in \mathbf{\Pi}$. Subsequently, function F can be decomposed as $F = gc + g'd$, where $g = a' + b$ serves as a control signal of the MUX.* □

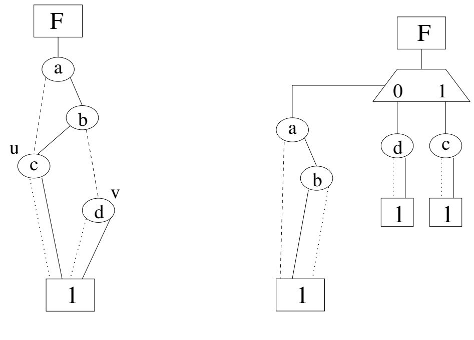


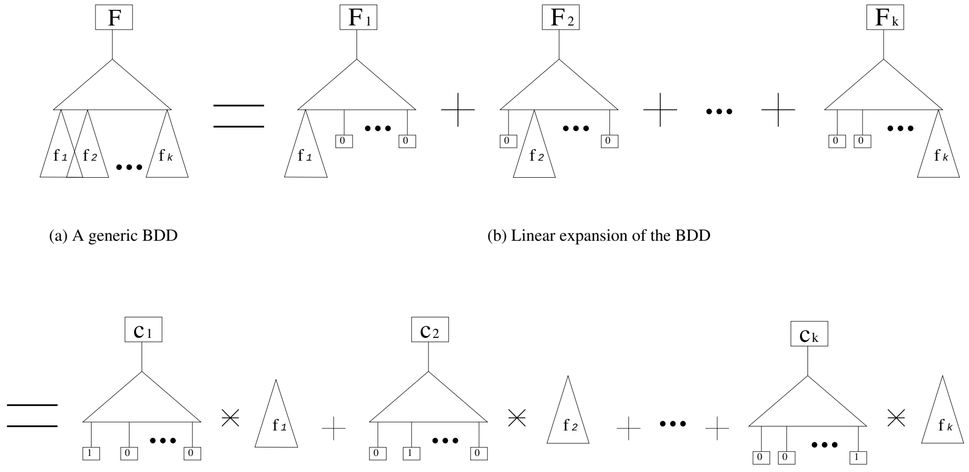
Fig. 21. Example of functional MUX decomposition: $F = gc + g'd$, $g = a' + b$.

F. Linear Expansion of BDDs

In this section, a generalization of different BDD decompositions described in previous sections is studied. It will be shown that all previous BDD decomposition methods are special cases of *linear expansion* to be presented here. Our objective is not to overrule all special-case BDD decompositions; these decompositions are of practical importance, because they are easy to identify and their decompositions are straightforward. The purpose of this section is to gain an understanding of the fundamentals of Boolean decomposition.

Fig. 22(a) shows a generic BDD. Each f_i represents an arbitrary logic function, including constant functions 0 and 1. Any BDD could be represented in this way without loss of generality. Let us examine the decomposition of such structured BDD into a set of disjunctive *component BDDs* $(F_1, F_2, \cdots, F_k)$, shown in Fig. 22(b). Each component BDD F_i consists of a *coefficient* BDD c_i and a *function* BDD f_i. Note that the root of each function BDD f_i plays a role of *1-dominator* in the respective component BDD. Therefore, each component BDD F_i can be further decomposed according to the 1-dominator structure. The final decomposition is shown in Fig. 22(c).

Now let us study the properties of *coefficient* BDDs $(c_1, c_2, \cdots, c_k)$. The relation between those coefficients are shown in Fig. 23. Since all coefficient BDDs are generated from the same BDD, and differ only in their terminals, all coefficient BDDs are graphically isomorphic. According to the principle of *APPLY* operation [15], the Boolean operations between those coefficient BDDs only take place at the terminals. Therefore, the union of all coefficient BDDs is equal to 1, which is shown in



(a) A generic BDD

(b) Linear expansion of the BDD

(c) Decomposition of all components using 1-dominator

Fig. 22. Linear expansion of a BDD

Fig. 23(a). Similarly, the intersection between any two different coefficient BDDs is equal to 0, which is shown in Fig. 23(b). Mathematically the above analysis can be formulated as follows.

Theorem 17 (Linear Expansion) *A Boolean function can be expanded w.r.t. an orthonormal coefficient set,* $c = \{c_1, c_2, \cdots, c_k\}$, *as follows:*

$$f = \sum_{i=1}^{k} c_i \; f_i$$

where $\sum_{i=1}^{k} c_i = 1$ *and* $c_i \cdot c_j = 0, \;\; \forall i \neq j$.

Proof: *Any Boolean function can be represented as a BDD with structure shown in Fig 22. This figure and Fig. 23 provide the proof.* □

We note that our *linear expansion* theory sounds exactly like the Definition 2 (orthonormal expansion). However, the way in which the two expansions are carried out is different. When a Boolean function is represented symbolically, in order to perform the orthonormal expansion, an orthonormal set must be provided first. Generation of such a symbolic orthonormal set is not trivial. Also the generalized co-factors required for the orthonormal expansion need to be calculated. Worst of all, the effectiveness of a symbolic orthonormal expansion will not be fully recognized until the whole decomposition is completed. In contrast to that, the linear expansion can be performed easily on a BDD, because the coefficient BDDs c_i and function BDDs f_i are represented explicitly by a BDD structure. The only thing that needs to be done is to figure out which set of coefficients should be used for the decomposition. Similarly, BDD structure provides lots of hints for this type of decomposition; some structural analysis of a BDD is required for this purpose. The effectiveness of linear expansion can also be readily estimated by the analysis of the BDD structure.

In summary, Theorem 17 provides further flexibility to decompose an arbitrary BDD. The applicability of this theorem relies on finding a BDD structure to which this theorem can be applied efficiently. The special cases, namely the *1-dominator*, *0-dominator*, *x-dominator*, *simple MUX*, and *functional MUX* decomposition, in which the number of component BDDs is limited to 2, have been taken care of in the previous sections. The structures more general than previously defined *dominators* should be identified. We anticipate that this generalization will further improve the performance of our BDD decomposition scheme.

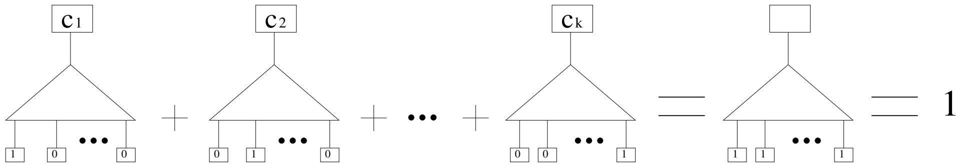
V. Logic Synthesis Based on BDD Decomposition - BDDlopt

In this section, implementation details of the logic optimization program, *BDDlopt*, which is based on our BDD decomposition theory, are presented. Algorithmic analysis of procedures in the proposed logic synthesis flow is also provided.

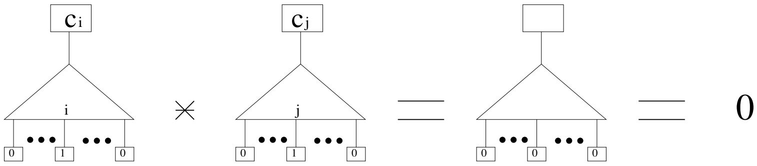
It will be shown that all necessary procedures in a typical logic optimization flow can be implemented through a series of BDD manipulations and decompositions. For example, Boolean simplification can be efficiently carried out through BDD variable reordering; factorization can be done through recursive BDD decompositions; and logic sharing can be efficiently detected on the final factoring trees.

A. Synthesis Flow

The synthesis flow for *BDDlopt* is outlined in Fig. 24. The flow consists of two major parts, BDD decomposition and factoring tree processing. First, the global BDDs (see Section II-C) are constructed for the Boolean network. Then, the global BDDs are submitted to the decomposition engine for logic decomposition. Along with the BDD decomposition, a set of factoring trees are constructed to record the decomposition. In the process of BDD decomposition, a large BDD is recursively decomposed into smal parts. The decomposition process stops when



(a) Sum of all coefficients equals 1.



(b) Intersection between any two different coefficients equals 0.

Fig. 23. Coefficient properties

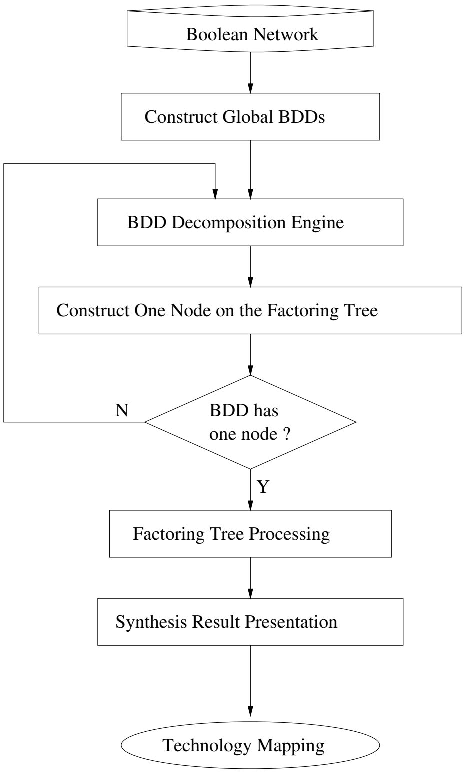


Fig. 24. Synthesis flow of `BDDlopt`

a BDD has one node. Finally, an important procedure, sharing extraction, takes place in the factoring tree processing phase.

Because of its iterative nature, the overall complexity is difficult to characterize. In the experiment, we will focus on the run time comparisons with the state-of-the-art logic synthesis program, SIS.

B. BDD store/load Mechanism

In this section, a BDD manipulation technique, which is crucial to performing logic simplification in *BDDlopt*, is explained.

In our BDD-based logic optimization scheme, BDD variable reordering algorithm serves as an implicit logic simplification. It should be emphasized that, in a typical BDD package, variables are reordered with respect to a BDD manager, and not w.r.t. a specific BDD. Hence, if there is more than one BDD in the manager, variable reordering may not result in the desired simplification for a specific BDD. In order to achieve maximum logic simplification of a Boolean function (BDD), all other BDDs must be freed from this BDD manager before performing variable reordering. However, those freed BDDs must be present in the BDD manager when they are needed for decomposition at a later time. Therefore, an efficient store/load mechanism must be developed.

A naive way of storing a BDD is to dump it into a SOP form. The advantage of SOP form is that the BDD can be reconstructed under a variable order which is different from the order in which the BDD is stored. This offers some flexibility for the implementation. However, since the number of SOP terms of a BDD can be exponential in the number of BDD nodes, storing BDDs in SOP form is not a feasible solution.

A new data structure, `bddPool`, has been devised to perform BDD store/load operations. Basically, a `bddPool` is a DAG which is graphically isomorphic to the BDD it represents. A BDD is copied to a `bddPool` before it is freed from the BDD manager. The BDD can be reconstructed later by applying an *ite*[8] operation n times, where n is the number of BDD nodes. Since an *ite* operation takes constant time, the overall complexity of our BDD store/load algorithm is $O(n)$. The disadvantage of `bddPool` is that the variable order of the BDD manager into which a BDD is loaded must be the same as the order in which a BDD is stored. Forcing a BDD manager to a certain variable order could result in an exponential increase in the BDD size if the manager is not empty.

[8] An *ite* is short for *if-then-else*; it is defined as $ite(x, g, h) = xg + x'h$

However, in our application, when a BDD is loaded (built), the BDD manager is always empty.

Another important feature of our `bddPool` mechanism is to allow the variable substitution during the process of BDD reconstruction. This can be accomplished easily by modifying the *ite* operator as $f = ite(\mathcal{M}(x), g, h)$, while $\mathcal{M}$ is the mapping of variables. This feature plays a crucial role in our efficient *iterative eliminate paradigm* (Section VI-C).

C. The BDD Decomposition Engine

Shown in Algorithm 1 is the main procedure for the BDD decomposition. To make the BDD manager "clean", all BDDs are stored in the `bddPool` form. A BDD is loaded into the BDD manager before it is decomposed. The store/load process is realized by function `storeBdd` and `loadBdd`. After a BDD f has been constructed in the BDD manager, it is decomposed by `decomposeBdd`. The decomposition results are presented as $g \odot h$, where $\odot$ stands for a Boolean operator, such as AND, OR, XOR or XNOR. The decomposition is stored in the form of factoring tree, discussed in Section V-D. The intermediate BDDs of g and h are then stored in the `bddPool` form and enqueued if they have more than one node. The decomposition process is iterated until the queue is empty.

```
bddPool = storeBdd(bdd);
Enqueue(Q, bddPool);
while(bddPool = Dequeue(Q)) {

   f = loadBdd(bddPool);
   (g, h, op) = decomposeBdd(f);

   construct one node on factoring tree;

   if (g != single node) {
      gPool = storeBdd(g);
      Enqueue(Q, gPool)
   }

   if (h != single node) {
      hPool = storeBdd(h);
      Enqueue(Q, hPool);
   }
}
return (factoring tree);
```

Algorithm 1: Main BDD decomposition flow

The main BDD decomposition engine, `decomposeBdd`, is a search process for the most efficient BDD decomposition, from more efficient (*algebraic*) to less efficient (*Boolean*). The dominators are empirically ordered in terms of decomposition efficiency as follows: 1) *simple dominator* (*1-, 0- and x-dominator*), 2) *functional MUX*, 3) *single MUX*, 4) *generalized dominator* and 5) *generalized x-dominator*. Finally, if all searches fail, the BDD is decomposed using *cofactor* w.r.t. the top variable. In practice, the last step is rarely reached. It is put here to ensure the BDD f will 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 is obtained. The information is used as a guidance for all the following decompositions. In terms of criticality of computational complexity, function `bddScan` is the most important one, because it is called every time the decomposition engine is invoked. The technique developed for this purpose is based on *edge marking*. The complexity of `bddScan` is $O(n|V|)$, where n is the number of BDD nodes, $|V|$ is the number of variables in the BDD.

C.1 Simple Dominator

All three simple dominators (*1-, 0-* and *x-dominator*) share a similar pattern, i.e., there is a node into which all internal edges converge. Based on this observation, efficient algorithms can be designed to unveil all simple dominator structures. In fact, the BDD scan procedure `bddScan` is devised to find out the structural information of a BDD. The structures of simple dominators are already encoded in the data collected by `bddScan`. The complexity of this function is $O(|V|)$.

In implementation, instead of returning the first found simple dominator, all simple dominators are obtained and the one closest to the middle height of the BDD is returned. This helps to achieve a more balanced decomposition, which is crucial to the delay minimization.

C.2 Generalized Dominators and Generalized x-Dominators

If an algebraic decomposition does not exist for a Boolean function, Boolean decomposition will be performed. The BDD structures leading to Boolean decompositions are *generalized dominators* and *generalized x-dominators*. Unlike the decompositions based on simple dominators, whose decomposition results are well-defined, the decompositions based on *generalized dominators* rely on BDD minimization w.r.t. a *don't care*. Therefore, the decomposition result depends on the efficiency of the BDD *don't care* minimization algorithms.

To carry out these decompositions, all possible Boolean decompositions are examined level by level. BDD scan information is required for the application of various filters. On each level, two major steps, *generalized dominator* generation and BDD minimization w.r.t. don't care, are involved in a single decomposition. The generation of a *generalized dominator* is a process of copying the BDD structure above the *cut*. The upper bound for this operation is $O(n)$. The function used to calculate f/d (or f/d') is based on *RESTRICT* operator [34] whose complexity is $O(|f|\cdot|d|)$ (or $O(|f|\cdot|d'|)$). The upper bound for *RESTRICT* is $O(n^2)$. Therefore, upper bound for function `decompGeneralizedDominator` is $(|V|n^2)$.

D. Construction and Processing of Factoring Trees

A factoring tree is a way to record a BDD decomposition process. For example, if a Boolean function f is decomposed into $g + h$, then a new node, with operator "+" and two siblings, g and h, will be created to record this decomposition. A factoring tree will keep growing until the BDD decomposition is completed. Subsequently, several steps can be applied to the factoring trees to further optimize the synthesis results. In particular, sharing between different factoring trees can be efficiently detected.

To identify the sharing between different factoring trees, BDDs are constructed for all factoring trees in a bottom-up fashion. The canonicity property of BDD is used to identify functionally equivalent sub-trees. Fig. 25 shows an example of sharing extraction on test case *b1.blif* from MCNC benchmark set.

E. Experimental Results

The experiments were conducted on SUN UltraSPARC-5/320M. They cover most of the combinational test cases from the MCNC benchmark set. All the test cases can be roughly categorized into two groups: 1) AND/OR-intensive functions, and 2) XOR-intensive logic (arithmetic functions). The literal count for decompositions generated by *BDDlopt* was compared with the number of literals in the factored form obtained by SIS-1.2 running *script.rugged*. The comparison also includes results after technology mapping. Both tree-based SIS mapper and Boolean matching-based *ceres* [35] are used. *ceres* is based on Boolean matching rather than tree matching. For this reason the XOR decompositions found by *BDDlopt* are likely to be preserved.

The results for AND/OR intensive circuits are shown in Table I. On average, *BDDlopt* uses slightly fewer gates than SIS, and more area than SIS. The slight increase in area is due to the higher cost of XOR gates implemented in CMOS. On average, the final synthesis results using *BDDlopt* and SIS on this class of functions are almost the same. While near optimal results are obtained by both SIS and *BDDlopt*, but *BDDlopt* out-performs SIS dramatically in CPU time. However, for the class of arithmetic functions and XOR-intensive logic, shown in Table II, *BDDlopt* outperforms SIS in all aspects. While, in principle, *ceres* generates better mapping results that SIS mapper, it was not stable on several circuits which makes the complete comparison difficult. For this reason only results of SIS mapper are presented. The results of techniques targeting specifically XOR decomposition by Tsai *et al* [36] are also listed for comparison purpose. One can see that the performance of *BDDlopt* in terms of the number of gates is comparable to that of Tsai *et al.* [36]. It should be noted that many XORs in the netlist synthesized by *BDDlopt* are lost after technology mapping. As indicated in column *XORs* in Table II, only 33% XORs are preserved in technology mapping.

VI. BDD-Based Logic Synthesis System - BDS

A very important feature of a logic synthesis system is its scalability. The scalability requires that the size of the *representation* of a *problem* be proportional to the size of the problem itself. In our case, the size of a BDD should be proportional to the size of a circuit (which is commonly measured by the number of logic gates). However, the size of global BDDs for a given Boolean network is completely unpredictable. It strongly depends on the type of the circuit, rather than on the total number of gates. Representing the entire Boolean network by global BDDs causes serious computational problems. Therefore, proper partitioning of the Boolean network is required prior to performing the BDD decomposition. Table III shows the comparison of the size of global BDDs and local BDDs (defined in Section II-C). It can be found that the size of global BDDs could be as much as two orders of magnitude larger than local BDDs.

The similar problem, large two-level representation, has also been observed in traditional multi-level logic synthesis. Fortunately, a proper way to handle it has been found. Given a large Boolean network, its multi-level structure should be preserved as much as possible. The number of SOP terms could be too large for the logic optimization algorithms if the entire Boolean network is collapsed into two-level forms. From this point of view, the network partitioning faced by BDD-based logic synthesis is similar to the one faced by traditional multi-level logic synthesis.

Circuits	Global BDDs	Local BDDs
C1355	33450	893
C1908	6734	1229
C2670	5554	1712
C3540	25828	2326
C432	1226	283
C499	26890	341
C5315	2942	3516
C7552	19322	5012
C880	15004	601
pair	4940	1808
rot	7340	934

TABLE III
COMPARISON OF NUMBER OF BDD NODES FOR GLOBAL AND LOCAL CONSTRUCTION

In this section, a new logic synthesis system, BDS, which has the capability to optimize arbitrarily large circuits, is presented.

A. Synthesis Flow

Current multi-level logic synthesis flow exemplified by SIS has drawn from over twenty years of intensive research. We believe it has the capability to handle very large circuits and it does grasp the essence of logic synthesis in general. Therefore, *BDS* adopts the general synthesis flow of SIS. Fig. 26 compares the synthesis flow of SIS and BDS. The similarity between them is obvious. The fundamental difference between SIS and BDS is the way in which each system represents Boolean

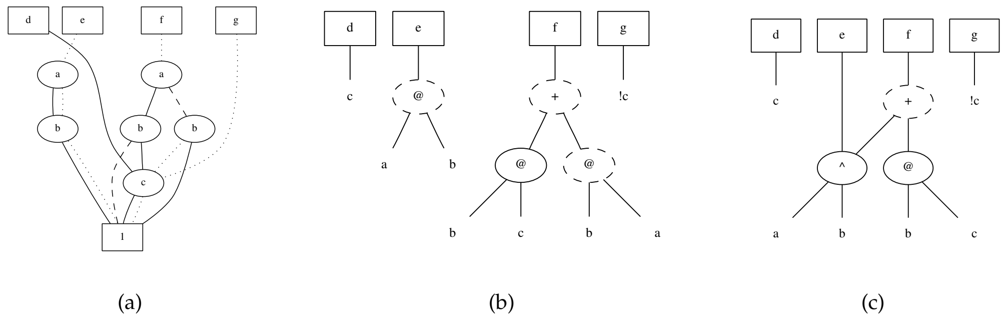


Fig. 25. Sharing extraction through the BDD construction. (a) Original BDD. (b) Factoring trees after BDD decomposition. (c) Factoring trees after sharing extraction.@ = XNOR; ∧ = XOR; *dashed* = negation.

Circuits			SIS				BDDlopt			
Name	In	Out	Lit.	gates	area	CPU	Lit.	gates	area	CPU
b1	3	4	10	5	144	0.2	9	4	128	0.0
b12	15	9	151	83	2384	19.4	77	45	1424	0.4
b9	41	21	122	92	2600	2.1	148	77	2328	1.3
c8	28	18	139	90	2440	1.9	140	75	2288	0.6
cc	21	20	58	32	920	1.1	74	46	1368	0.4
cht	47	36	165	48	2328	1.9	193	110	3008	1.2
cm138a	6	8	31	17	472	0.9	31	15	488	0.1
cm150a	21	1	51	21	720	0.5	53	38	1200	0.5
cm151a	12	2	26	17	528	0.4	26	15	424	0.3
cm152a	11	1	22	16	512	0.2	22	13	360	0.1
cm162a	14	5	49	26	816	0.7	52	27	872	0.2
cm163a	16	5	49	31	832	0.7	37	20	672	0.1
cm42a	4	10	34	17	472	0.8	35	17	552	0.1
cm82a	5	3	24	9	296	0.2	16	9	336	0.1
cm85a	11	3	46	28	824	0.6	43	32	960	0.1
cmb	16	4	51	27	880	0.4	39	14	592	0.2
con1	7	2	20	13	368	0.2	21	12	368	0.1
count	35	16	143	96	2680	2.0	159	77	2824	1.4
cu	14	11	60	35	1016	1.0	72	35	1192	0.3
decod	5	16	52	31	840	1.1	60	30	824	0.2
frg1	28	3	136	107	3280	8.3	102	56	1760	1.3
majority	5	1	10	6	200	0.2	10	5	184	0.1
misex2	25	18	106	65	1832	1.3	177	87	3016	0.7
o64	130	1	-	-	-	-	130	80	2312	2.4
pcle	19	9	69	44	1256	0.9	77	51	1560	0.4
pm1	16	13	50	30	800	0.8	64	27	896	0.2
sct	19	15	79	48	1328	2.0	83	48	1488	0.4
tcon	17	16	32	9	400	0.3	40	24	576	0.1
ttt2	24	21	217	138	3952	5.9	201	121	3928	1.1
unreg	36	16	102	52	1512	1.5	130	66	1952	0.8
Total			2104	1233	36632	57.5	2814	1196	37568	12.8
Average Ratio (BDDlopt/SIS)							137%	104%	105%	37%

TABLE I

AND/OR-INTENSIVE CIRCUITS: RESULTS OF LOGIC OPTIMIZATION WITH *BDDlopt-1.2.5* VS SIS. TECHNOLOGY MAPPING IS DONE BY *ceres*. CIRCUITS ARE MAPPED TO LIBRARY *msu cmos3*.

Circuits	SIS				BDDlopt					Tsai [36]
Name	Lit.	gates	area	CPU	Lit.	gates	area	CPU	XORs	gates
5xp1	132	81	195	4.1	95	67	172	0.4	4/16	66
9sym	274	152	396	22.0	70	42	109	1.0	0/4	64
9symml	186	102	270	19.7	70	41	108	0.9	0/4	-
alu2	361	217	524	74.7	318	230	632	2.8	13/53	-
alu4	694	409	996	286.3	930	582	1655	15.9	23/124	-
cordic	64	34	94	0.9	56	47	126	0.5	6/16	-
f51m	98	58	139	9.0	73	56	174	0.3	5/11	63
my_add	192	156	287	3.1	128	110	286	8.9	16/32	113
parity	60	15	75	0.6	16	15	75	0.1	15/15	15
rd53	34	22	47	1.3	38	25	72	0.2	3/6	25
rd73	189	106	258	12.1	80	45	133	0.8	5/8	41
rd84	348	192	468	42.8	115	62	189	1.4	6/12	66
t481	881	407	1023	208.6	16	15	45	0.3	5/5	23
z4ml	41	20	59	2.2	24	20	53	0.1	3/6	21
Total	3554	1971	4831	687.4	2029	1357	3941	33.6	104/312	
Average Ratio (BDDlopt/SIS)					60%	77%	86%	15.6%	33%	

TABLE II
XOR-INTENSIVE CIRCUITS: RESULTS OF LOGIC OPTIMIZATION WITH *BDDlopt-1.2.5* VS. SIS AND TSAI [36]. SINCE *ceres* IS NOT STABLE ON THIS CLASS OF FUNCTIONS, ONLY SIS MAPPER IS USED. CIRCUITS ARE MAPPED TO *mcnc.genlib*. THE NUMBER OF XORS AFTER/BEFORE TECHNOLOGY MAPPING IS SHOWN IN COLUMN *XORs*.

nodes and carries out all individual synthesis procedures. In *BDS*, after a Boolean network has been built, all Boolean nodes are represented as local BDDs. All the following procedures are carried out based on the local BDDs.

Although the space for further improvement of the synthesis flow seems to be limited, there is still a potential for significant improvement in its many procedures [3]. This is especially true for our BDD-based logic synthesis, in which all procedures are formulated in the BDD domain. It should be mentioned that all procedures in the synthesis flow are heavily influenced by the underlying Boolean representation. Logic representation based on BDDs is significantly different from traditional SOP forms. Therefore, while retaining a similar synthesis flow, new algorithms, specially tailored for BDDs, have been developed for all the procedures.

In the following sections, essential procedures in the synthesis flow are reviewed, and the corresponding implementations in the BDD domain are presented.

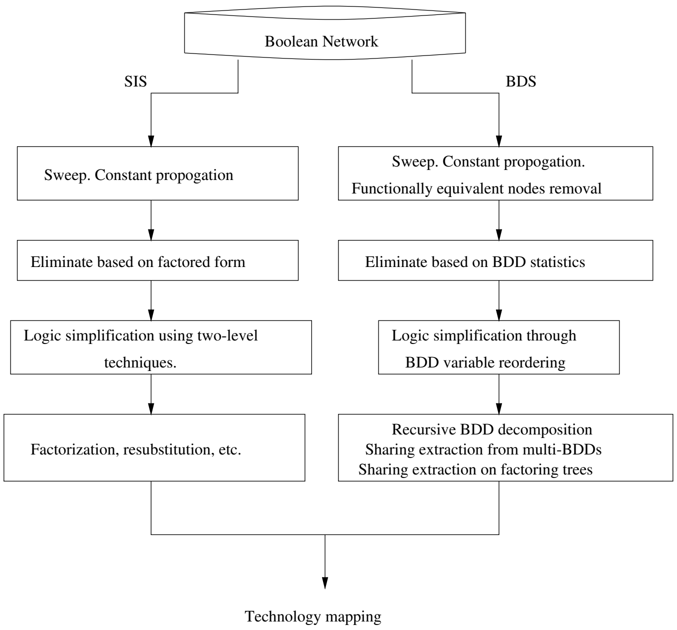


Fig. 26. Synthesis flow of SIS and BDS

B. Sweeping Operation

`sweep` is the first step in the proposed synthesis flow. It removes some obvious redundancy from the Boolean networks. Although there is no real logic optimization involved in this procedure, for certain multi-level Boolean networks, `sweep` plays an important role in removing redundancy from the networks.

B.1 Constant and Single-Variable Nodes Removal

Constant nodes in a Boolean network are caused by the way a Boolean network is represented. For example, a primary input might be connected to the ground or power permanently. A logic optimization program should take advantage of constant and single-variable nodes to reduce the complexity of a Boolean network. It should be noted that the removal of one constant or single-variable node may create another, and such nodes may be produced during the process of logic optimization. Therefore, `sweep` is iterative.

B.2 Removal of Functionally Equivalent Nodes

Traditionally, when a multi-level Boolean function is represented as a cube form, only constant and single-variable nodes can be identified and removed from the Boolean network during `sweep`. When a Boolean network is represented in the BDD form, there is an added bonus. Since BDD is canonical, the functional equivalence between different Boolean nodes can be detected easily. Therefore, in addition to removing constant and single-variable nodes, all functionally equivalent nodes can be also be removed from a Boolean network during `sweep`.

Although the functionally equivalent Boolean nodes in a Boolean network can also be removed by later optimization procedures (*e.g. eliminate, re-substitution*) in the traditional synthesis flow, it is always beneficial to remove such redundancy before the actual logic optimization. Table IV lists the number of functionally duplicated nodes for some well-known testcases. We were surprised to find so much redundancy in those testcases. This is the first time ever that functionally duplicated Boolean nodes can be removed before actual logic optimization.

Circuits	Total Nodes	Duplicated Nodes
C1908	441	118
C2670	787	72
C3540	956	247
C5315	1467	197
C6228	2353	30
C7552	2165	355
C880	302	10
dalu	985	249
i8	1183	186
i9	329	22
i10	1634	84
pair	830	16
vda	123	3

TABLE IV
NUMBER OF FUNCTIONALLY DUPLICATED NODES IN A BOOLEAN NETWORK

Since the removal of one functionally equivalent node may create another, the duplication removal in *BDS* is iterative. The numbers shown in Table IV are just the numbers of duplicated nodes found in the first iteration. The actual number is even larger. Removing functionally duplicated Boolean nodes helps *BDS* to reach the final optimized netlist. This also contributes to the runtime advantage over traditional approaches, because logic optimization algorithms are generally more expensive than `sweep`.

C. Boolean Network Partitioning by Iterative Node Elimination

Due to the size of most industrial designs and the limited computational capacity of logic optimization algorithms, it is not practical to apply logic optimization algorithms on global representations. It can also be shown that, on the other extreme, applying logic optimization algorithms to completely local representations does not work either. In a typical logic synthesis flow, since a Boolean network is obtained through the direct translation from HDL languages (Verilog, VHDL, *etc.*), most components of the network are simple gates. Therefore, it is an overkill to apply logic optimization algorithms on simple gates. A reasonable trade-off in this *global vs. local* scenario should be found. A Boolean network should be allowed to be partially collapsed into a set of *super* Boolean nodes and each represented as a BDD; then logic optimization algorithms can be applied to each super node. It is obvious that the procedure to carry out the partial collapsing is critical to a logic synthesis system.

In multi-level logic synthesis, in addition to providing a trade-off between local and global representations, partial collapsing also helps to remove logic redundancy embedded in a multi-level configuration. The most frequent cause of redundancy in a multi-level Boolean network is the so-called *re-convergence*. This type of redundancy can be easily removed by partial node collapsing.

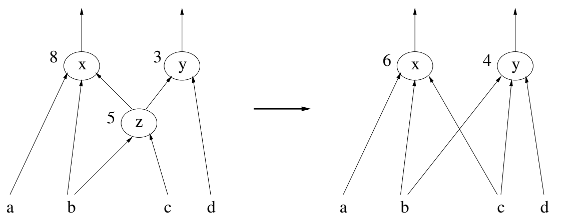
To carry out partial collapsing, procedure such as `eliminate` comes into play. `eliminate` attempts to find a partially collapsed Boolean network such that Boolean nodes are not too large for logic optimization algorithms. On the other hand, Boolean nodes should not be too fine, otherwise some redundancy may remain in the network. A properly designed `eliminate` scheme will provide a better starting point for logic optimization algorithms.

`eliminate` has been successfully implemented in SIS [2], in which an implementation cost is associated with each Boolean node. The decision whether a Boolean node is collapsed into its fanout depends on the cost gain, measured as a difference in the number of BDD nodes before and after an attempted collapse of the Boolean node into its fanouts.

Two approaches have been proposed for `eliminate` through BDD manipulation. The first one is based on *progressive* elimination [37]. In this approach, BDDs are constructed from primary inputs to primary outputs. At any point, if the size of a BDD is larger than a pre-defined threshold, an intermediate variable is introduced and the BDD construction process continues until primary outputs are reached. This approach ignores the specific structure of a Boolean network. As a result, the elimination often stops at boundaries which are not natural to the specific Boolean network. This approach may also cause memory blow-up. The second approach is based on *iterative* elimination [38] which is quite similar to the `eliminate` procedure in SIS [2]. In the process, BDD *node count* is used as the cost function to guide the elimination.

To comply with the mainstream synthesis flow, an approach similar to that of SIS [2], [38] has been adopted. However, due to the efficiency of described techniques for BDD manipulations, our `eliminate` is orders of magnitude faster than [38].

Example 8: *Fig. 27 shows an example of node elimination. The initial Boolean network is shown in Fig. 27(a). The number shown next to each Boolean node is the cost (number of BDD nodes) associated with the Boolean node. The cost of the initial network is 16. The BDD of node z is composed into the BDDs of x and y. The new cost is 10. The difference between new and initial cost is -6. If the difference is less than a pre-specified threshold, node z is eliminated from the network and the inputs of x and y are modified accordingly, as shown in Fig. 27(b). If the difference is larger than the threshold, the network will be left unchanged.* □



(a) Initial Boolean network (b) Network after node elimination

Fig. 27. An example of node elimination.

A generic algorithm for iterative elimination is shown in Algorithm 2. In `collect`, all Boolean nodes which are eligible for collapsing into their fanouts are collected. In `execute`, the Boolean network is modified to lock the recent changes. The BDD manager is reordered before the next `collect`. The whole process is iterated until no Boolean node can be eliminated any further.

```
candidate = collect(bddmgr,network);
while(candidate) {
   execute(bddmgr,candidate);
   reorder(bddmgr);
   candidate = collect(bddmgr,network);
}
```

Algorithm 2: Eliminate

In practice, a straightforward implementation of this process is not efficient. This is mainly because of the abuse of 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 huge even for a medium-size circuits. Since the complexity of variable reordering could be exponential, reordering a BDD manager with large number of variables will severely degrade the overall runtime performance.

In *BDS*, new BDD manipulation techniques are developed to make the approach feasible in practice. Algorithm 3 shows the modified version of Algorithm 2. In this algorithm, instead of using a single BDD manager for all the operations, a new BDD manager is initialized in each iteration. All BDDs after one iteration are *mapped* into the

```
candidate = collect(bddmgr,network);
while(candidate) {
   execute(bddmgr,candidate);
   newbddmgr = bddMapping(bddmgr,network);
   reorder(newbddmgr);
   free(bddmgr);
   bddmgr = newbddmgr;
   candidate = collect(bddmgr,network);
}
```

Algorithm 3: Efficient Iterative Eliminate Paradigm

new BDD manager. A variable reordering is performed for the new BDD manager. The old BDD manager is abandoned. The process is iterated until no Boolean nodes can be eliminated.

The need for a new BDD manager and the BDD *mapping* operation can be justified as follows. A typical approach for variable reordering is based on adjacent variable swapping. To find the optimal position for a variable, a bulk of *unique table* of a BDD manager will be traversed. During the process of *eliminate*, the removal of one Boolean node from the Boolean network corresponds to the demise of one variable in the BDD manager, so the variable will not be used again. Let us refer to these variables as *unused* variables. After the termination of one iteration many Boolean nodes have been removed, and the BDD manager contains large number of *unused* variables. Table V shows the reduction in the number of Boolean nodes after first iteration. It can be found that about 63% variables in the BDD manager become *unused*. It is obvious that performing variable reordering in a BDD manager with large number of *unused* variables is very inefficient.

Circuits	Before	After	Reduction
C1355	474	60	88 %
C1908	325	94	72 %
C2670	656	281	58 %
C3540	793	344	57 %
C432	123	63	49 %
C499	162	57	65 %
C5315	1228	387	69 %
C6288	2338	704	70 %
C7552	1829	455	76 %
C880	296	122	59 %
dalu	764	241	69 %
des	681	294	57 %
mult32	5507	2467	56 %
pair	818	450	45 %
Total	15994	6019	63 %

TABLE V

NUMBER OF BOOLEAN NODE REDUCTION AFTER FIRST *eliminate*

Instead of reordering the BDD manager with large number of *unused* variables, a new BDD manager with set of *used* variables is initialized. BDDs are then transfered into the new BDD manager using our `bddPool` mechanism (see Section V-B). During the process, variables are substituted according to $\mathcal{M}$, where $\mathcal{M}$ is the mapping of variables between the old and new BDD managers. When all BDDs are reconstructed in the new BDD manager, a set of BDDs which are graphically isomorphic to the original ones, but much more compact in the range of indices, are obtained. This process is referred to as a BDD *mapping*.

Table VI shows the results of our *iterative eliminate paradigm*. The results of Chaudhry [38] are also listed for comparison. On average, our `eliminate` is 85 times faster than [38]. The runtime advantage of *BDS* becomes stronger for larger test cases. Although *BDS* is targeting multi-level implementation, it is obvious that the iterative eliminate paradigm in *BDS* will also be useful for PTL synthesis. Due to an efficient way of handling BDD variable reordering, our iterative eliminate paradigm has the capability to handle arbitrarily large circuits.

D. Experimental Results

The experiments have been conducted on a Pentium-III/500MHz machine running Linux. Most large combinational circuits in *MCNC* test case suite are covered in the experiment. All test cases are synthesized by both *BDS* and SIS (*script.rugged*). The results are then mapped by the SIS mapper.

Table VII shows the experimental results. Two testcases, *dalu* and *vda*, have been singled out from this table for special illustration. In summary, the synthesis results of *BDS* uses about 3% more area than that of SIS. The delay of circuits synthesized by *BDS* is 13% smaller. The memory required by *BDS* is 30% smaller than SIS. It should be noted that the memory usages reported for SIS in Table VII only include the memory used by logic optimization procedures in *script.rugged*. Memory used by `full_simplify` is not included. Since global BDDs are constructed during `full_simplify`, which is not used by *BDS*, it is unfair to compare the total memory used by *script.rugged* with *BDS*. In terms of runtime, *BDS* demonstrates superior advantage over SIS; it is more than 8 times faster than SIS. We must mention that compared with real industrial circuits, all the test cases used in this experiment are relatively small. To prove the potential of *BDS* to optimize large circuits, we run both *BDS* and SIS on a set of large circuits generated by a proprietary HDL to BLIF translator. The results are shown in Table VIII. It is obvious that the runtime of *BDS* is significantly lower than that of SIS.

There are two causes which contribute to the larger circuit area obtained by *BDS*. First is due to *BDS*'s capability to perform XOR and MUX decompositions. XOR and MUX operators have been represented explicitly on the factoring trees and in the final BLIF netlist generated by *BDS*. However, due to the weak capability of tree-based technology mapper to identify XORs and MUXes, only a small fraction of XORs and MUXes synthesized by *BDS* can be mapped to XOR and MUX gates. The same problem has been observed in our previous experiment [39].

Second, currently *BDS* does not have the capability of simplifying a Boolean network by using the *don't cares* embedded in a multi-level configuration. If the redundancy can not be removed by `eliminate`, it will most likely remain in the final synthesized circuits. Lack of such capability is the major hold-back for the current version of *BDS*. Shown in Table IX are the synthesis results for circuits *dalu* and *vda*. The results can be greatly improved by applying `full_simplify` on the circuits synthesized by *BDS*. However, the area and delay of circuit *dalu* is still 50% more than SIS. Extensive comparison between *BDS* and SIS should be done for *dalu*.

VII. Conclusions

In this paper, a BDD-based logic optimization system, based on a new BDD decomposition theory, is presented. The new BDD decomposition theory has a great potential to significantly improve the existing logic optimization methods. A new logic optimization system, *BDS*, has been successfully developed. Detailed implementation of the overall synthesis strategy, including network partitioning by partial node collapsing, BDD-based Boolean decomposition, factorization, and sharing extraction have been presented in the paper. The experimental results clearly demonstrate that *BDS* has a superior runtime advantage over traditional approaches.

The capacity of current *BDS* can be further enhanced by incorporating the following future work:

1. A general BDD decomposition should be developed based on *linear expansion* theory.
2. BDD-based *don't care* minimization, similar to *full_simplify* in SIS, should be developed.
3. Recently, we found that *BDS* is also amenable to FPGA synthesis. In-depth analysis of the synthesis results of *BDS* should be performed to understand the reason for its applicability to FPGAs. We anticipate that our BDD-based logic optimization will also be applicable to FPGA synthesis. Very encouraging initial results have been already obtained [40].

Compared with the state-of-the-art logic synthesis methodology, which has evolved from continuous research and development for the last twenty years, the BDD-based logic synthesis is brand new but much less mature. Extensive fundamental research has to be done to make this approach a truly successful synthesis method. It is too soon to conclude whether BDD-based logic synthesis approach will become a practical alternative to the widely accepted traditional methods. We hope this research will initiate a new round of research in logic synthesis area in the years to come.

References

[1] R.K. Brayton, G.D. Hachtel, and A. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," in *Proc. IEEE*, Feb. 1990, pp. 264–300.

Circuits	Chaudhry *et al* [38]		BDS	
	BDD nodes	CPU (s)	BDD nodes	CPU (s)
C1355	211	270	207	0.3
C1908	310	25.4	276	0.6
C2670	615	197	527	1.7
C3540	974	101.7	901	3.2
C432	181	4.5	183	0.5
C499	196	2.4	228	0.2
C5315	1008	307.6	918	4.0
C6288	1677	540.7	1507	4.4
C7552	1592	382.1	1227	6.4
C880	298	7.5	300	0.4
Total	7066	1838.9	6274	21.7

TABLE VI

RESULTS OF ITERATIVE ELIMINATE PARADIGM. BOTH EXPERIMENTS ARE CARRIED OUT ON PENTIUM-200.

Circuits	SIS				BDS			
	Area	Delay	CPU (s)	Mem	Area	Delay	CPU (s)	Mem
C1355	689	39.40	6.6	1.2	711	45.60	0.4	1.0
C1908	695	68.60	8.1	1.2	730	65.00	0.8	1.0
C3540	1695	81.40	16.1	3.3	1713	81.20	3.6	1.9
C432	290	75.90	46.1	0.7	357	78.40	0.2	0.5
C499	689	39.40	6.8	0.9	708	43.60	0.6	0.5
C5315	2286	68.60	10.2	3.1	2402	70.50	5.3	3.0
C6288	4631	237.8	21.8	4.1	4677	178.3	3.8	1.1
C7552	3038	115.70	54.2	4.9	3112	83.30	4.2	4.8
C880	567	56.10	1.9	1.0	563	43.20	0.7	0.8
pair	2274	74.30	16.1	2.5	2466	52.60	2.1	2.0
rot	965	51.60	4.5	2.0	1025	51.90	1.0	0.9
Total	17819	908.8	192.4	24.9	18464	793.6	22.7	17.5

TABLE VII

COMPARISON BETWEEN BDS AND SIS. THE MEMORY REPORTED FOR SIS DOES NOT INCLUDE THE MEMORY USED BY `full_simplify`.

Circuits	SIS				BDS				Speed
Name	gates	cost	delay(ns)	CPU(s)	gates	cost	delay(ns)	CPU(s)	up
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	

TABLE VIII

RESULTS OF BDS AND SIS ON A SET OF LARGE CIRCUITS.

Circuits	SIS		BDS-1.3			
			no full_simplify		full_simplify	
	Area	Delay	Area	Delay	Area	Delay
dalu	1307	58.40	2680	103.5	1927	93.3
vda	837	39.8	1380	32.6	1049	43.2

TABLE IX

THE EFFECT OF `full_simplify`.

[2] E. Sentovich *et al.*, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley., 1992.

[3] R. Rudell, "Tutorial : Design of a logic synthesis system," in *Proc. 33rd Design Automation Conference*, 1996, pp. 191–196.

[4] R.K. Brayton, C. McMullen, G.D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[5] O. Coudert and J.C. Madre, "New ideas for solving covering problems," in *Proc. Design Automation Conference*, 1995, pp. 641–646.

[6] G Boole, *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*, 1854.

[7] C. Y. Lee, "Representation of switching circuits by binary decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, June 1959.

[8] S. B. Akers, "Functional Testing with Binary Decision Diagrams," in *Eighth Annual Conference on Fault-Tolerant Computing*, 1978, pp. 75–82.

[9] F. Brown, *Boolean Reasoning*, Kluwer Academic Publishers, Boston, MA, 1990.

[10] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Company, 1970.

[11] R.L. Ashenhurst, "The Decomposition of Switching Functions," in *Proc. of an International Symposium on the Theory of Switching*, Cambridge, MA, 1957, vol. XXIX of *The Annals of the Computation Laboratory of Harvard University*, pp. 74–116, Harvard University Press, Published in 1959.

[12] J.P. Roth and R.M. Karp, "Minimization Over Boolean Graphs," in *IBM J. Res. Dev.*, April 1962, pp. 227–238.

[13] H.A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Company, Inc, 1962.

[14] T. Singer, "Some uses of truth tables," in *International Symposium on the Theory of Switching, pt. I*, 1959, pp. 125–133.

[15] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computer*, vol. 35, no. 8, pp. 677–691, August 1986.

[16] K. Brace, R. Rudell, and R. Bryant, "Efficient Implementation of a BDD Package," in *Proc. Design Automation Conference*, 1990, pp. 40–45.

[17] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," in *Proc. Intl. Conf. on Computer-Aided Design*, 1988, pp. 2–5.

[18] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proc. Intl. Conf. on Computer-Aided Design*, 1988, pp. 6–9.

[19] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proceedings of the European Conference on Design Automation*, Amsterdam, 1991, pp. 50–54.

[20] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," in *Proc. Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 472–475.

[21] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in *IEEE International Conference on Computer-Aided Design*, 1993, pp. 42–47.

[22] M. Sauerhoff and I. Wegener, "On the Complexity of Minimizing the OBDD Size for Incompletely Specified Functions," *IEEE Trans. on CAD*, vol. 15, pp. 1435–1437, Nov. 1996.

[23] O. Coudert and J.C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. ICCAD*, 1990, pp. 126–129.

[24] T. Shiple, R. Hojati, A. Sangiovanni-Vicentelli, and R. Brayton, "Heuristic Minimization of BDDs Using Don't Cares," in *Proc. Design Automation Conference*, 1994, pp. 225–231.

[25] Youpyo Hong, Peter Beerel, Jerry Burch, and Kenneth McMillan, "Safe BDD Minimization Using Don't Cares," in *Proc. Design Automation Conference*, 1997.

[26] Shih-Chieh Chang, M. Marek-Sadowska, and T. Hwang, "Technology Mapping for TLU FPGA's Based on Decomposition of Binary Decision Diagrams," *IEEE Trans. on CAD*, vol. 15, no. 10, pp. 1226–1235, October 1996.

[27] Yung-Te Lai, Kuo-Rueih Pan, and Massoud Pedram, "OBDD-Based Function Decomposition: Algorithms and Implementattion," *IEEE Trans. on CAD*, vol. 15, no. 8, pp. 977–990, August 1996.

[28] M. A. Thornton and V. S. S. Nair, "Behavioral Synthesis of Combinational Logic Using Spectral Based Heuristics," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 219–230, April 1999.

[29] Kevin Karplus, "Using if-then-else DAGs for Multi-Level Logic Minimization," Tech. Rep. UCSC-CRL-88-29, University of California Santa Cruz, 1988.

[30] V. Bertacco and M. Damiani, "The Disjunctive Decomposition of Logic Functions," in *IEEE International Conference on Computer-Aided Design*, 1997, pp. 78–82.

[31] Ted Stanion and Carl Sechen, "Boolean Division and Factorization Using Binary Decision Diagrams," *IEEE Trans. on CAD*, vol. 13, no. 9, pp. 1179–1184, September 1994.

[32] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multiple-Level Logic Optimization System," in *IEEE Trans. on CAD*, June 1987, vol. 6, pp. 1062–1081.

[33] R. K. Brayton, ," University of California, Berkeley. Notes on Multi-level Logic Synthesis, 1988.

[34] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Sequential Machines Based on Symbolic Execution," in *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed., Grenoble, France, June 1989, vol. 407 of *Lecture Notes in Computer Science*, pp. 365–373.

[35] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on boolean operations," *IEEE Trans. on CAD*, vol. 12, no. 5, pp. 599–620, May 1993.

[36] C. Tsai and M. Marek-Sadowska, "Multilevel Logic Synthesis for Arithmetic Functions," in *Proc. Design Automation Conference*, 1996, pp. 242–247.

[37] P. Buch, A. Narayan, R. Newton, and A. Sangiovanni-Vincentelli, "On Synthesizing Pass Transistor Logic," in *Intl. Workshop on Logic Synthesis*, 1997.

[38] R. Chaudhry, T. Liu, A. Aziz, and J. Burns, "Area-Oriented Synthesis for Pass-Transistor Logic," in *International Conference on Computer Design*, 1998, pp. 160–167.

[39] C. Yang, V. Singhal, and M. Ciesielski, "BDD Decomposition for Efficient Logic Synthesis," in *International Conference on Computer Design*, 1999, pp. 626–631.

[40] Russell Tessier and Nemuri Navin, ," University of Massachusetts, Amherst. Personal communication, 1999.