BDD DECOMPOSITION FOR EFFICIENT LOGIC SYNTHESIS

Congguang(Anda) Yang Vigyan Singhal Maciej Ciesielski
cyang@illiac.ecs.umass.edu vigyan@cadence.com ciesiel@ecs.umass.edu

Abstract—
There are two major approaches to the synthesis of logic circuits. One is based on a predominantly algebraic factorization leading to AND/OR logic optimization. The other is based on classical Reed-Muller decomposition method and its related decision diagrams, which have been shown to be efficient for XOR-intensive arithmetic functions. Both approaches share the same characteristics: while one is strong at one class of functions, it is weak at the other's.

In this paper, we propose a unified logic optimization method which proves very efficient at handling both AND/OR-intensive and XOR-intensive functions. The proposed method is based on iterative BDD decomposition using various *dominators*. Detail analysis of decomposable BDD structures leading to AND/OR, XOR and MUX decompositions are presented. Experiment shows that our synthesis results for AND/OR-intensive functions are comparable to those of SIS, and results for XOR-intensive functions are comparable to those of techniques targeting XOR decomposition.

I. INTRODUCTION

Traditional logic synthesis methodology, based on algebraic transformation of Boolean networks [1], [2], has gained well deserved popularity in transforming two-level SOP forms into multi-level logic. While near optimal results can be obtained for those Boolean functions which can be compactly 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. Recently, efficient methods have been developed to handle AND-XOR decomposition of arithmetic functions [3], [4]. The main platforms used in those methods are decision diagrams or decomposition graphs which have inherent XOR decomposition of its nodes. While these methods generate much better multi-level representation for XOR-intensive functions, the results for functions with predominantly AND/OR logic do not compare favorably with those of SIS [2]. For example, circuit $i3$ (with 132 inputs and 6 outputs) from the MCNC benchmark set has a compact AND/OR factored form representation with 188 literals, while the optimized exclusive-SOP form requires 4,076,549 literals [4]. A method [5] is proposed to resolve this dilemma. The *fixed polarity RM* (FPRM) representation of a Boolean function is derived first. If the complexity of this representation is more than an empirical threshold, the function is handed over to the well-known AND/OR logic minimization tool MISII. However, the majority of random and control logic circuits exhibit a sporadic combination of AND/OR and XOR logic. Therefore good decompositions are hard to be found this way.

Recently there has been a growing interest to perform multi-level logic optimization directly on binary decision diagrams (BDD's) [6]. Compared with the traditional logic optimization platform (two-level SOP), using BDD to carry out logic optimization has the following advantages: 1) BDD is not susceptible to the structural variations. 2) Both algebraic and Boolean decompositions which lead to dramatic logic minimization are preserved on BDDs. Efficient algorithms have been developed in this paper to identify these decompositions. 3) It has been observed that variable reordering on a BDD performs some degree of logic optimization implicitly. Therefore, a reordered BDD provides a better starting point than a primitive structural netlist. We will comment on this more in the sequel. 4) The process of constructing a BDD removes some redundancy automatically.

Finding multi-level representation of Boolean functions through BDD decomposition was studied by Karplus [7] at the early ages of BDDs. Since then, 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 using BDDs. Bertacco *et al.* [8] 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 (9symml, for example). Based on our experience, their method fails to generate good decompositions on BDDs with complement edges. Stanion *et al.* [9] proposed a *generalized cofactor*-based Boolean division and factorization method. Given a divisor D, a function F can be written as $F = D \cdot cof(F, D) + D' \cdot cof(F, D')$. Therefore, 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 lack of an 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.

BDDs are constructed using *Shannon* expansion. Therefore, it is straightforward to obtain a Boolean expression which consists of only AND/ORs. It is relatively difficult to derive an expression which contains XORs. On the contrary, XOR decomposition can be easily identified on graphs [4] or decision diagrams [10], [11] obtained by applying *Davio* expansion. However, by using a technique called *complement edges removal*, we are going to show that BDDs can also be used to perform XOR decomposition efficiently.

To the best of our knowledge, there is no logic optimization method which can efficiently perform logic optimization on both AND/OR- and XOR-intensive functions. In this paper, we introduce a unified logic optimization methodology which proves very efficient at handling both AND/OR- and XOR-intensive functions. Furthermore, it also handles nontrivial MUX decomposition, not available by other methods. The proposed method is based on iterative decomposition of BDDs using various *dominators* discussed in the sequel. Our synthesis results for AND/OR-intensive functions are comparable to those of SIS, and the results on XOR-intensive functions are comparable to those of techniques targeting specifically at XOR decompositions. The following examples, taken from the MCNC benchmarks suite, will illustrate our claims.

Example 1: The above-mentioned circuit *i3*. We find that the circuit can be completely algebraically decomposed using AND and OR operators.

Example 2: circuit *t481* is a 16-input, single-output cir-

C. Yang and M. Ciesielski are with the Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, MA 01003. V. Singhal is with the Cadence Berkeley Lab, Berkeley, CA, 94704.

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

cuit. The circuit optimized by SIS-1.2 using *script.rugged* and mapped to *mcnc.genlib* library consists of 407 gates with total cost of 1023. On other hand, this circuit has only 16 literals when represented in *fixed polarity Reed-Muller* (FPRM) form [3]. Our tool successfully found the optimal (16-literal) factored form of the circuit, shown in the following equation.

$$\begin{aligned} t481 = {} & ((v0+\overline{v1})\overline{\oplus}(\overline{v2}+v3)+(v4+\overline{v5})\oplus(\overline{v6}+v7))\overline{\oplus} \\ & ((v8+\overline{v9})\oplus(\overline{v10}+v11)+(v12+\overline{v13})\overline{\oplus}(\overline{v14}+v15)) \end{aligned}$$

This form is different from that obtained in [3]. The mapped circuit based on this factored form consists of only 15 gates with the total cost of 45.

The rest of the paper is organized as follows. AND/OR decomposition on BDDs is described in Section II; XOR and MUX decompositions are presented in Section III. The implementation issues and results are presented in Section IV. Several related issues, such as the role of variable reordering in *BDDlopt*, BDD minimization, are discussed in Section V. All BDDs are drawn according to the conventions in CUDD [12]. *Solid* lines represent *1-edges*, *dashed* lines represent *0-edges* and *dotted* lines represent *complement edges.*

II. AND/OR Decomposition On BDDs

The main idea of AND/OR decomposition of a BDD was presented in [13]. For the sake of completeness we review here the basic definitions and theorems related to this type of decomposition. Proofs of the theorems are given in [13]. Readers not familiar with BDDs are referred to [6].

A BDD is a directed acyclic graph (DAG) representing a Boolean function. It can be uniquely defined as a tuple, **BDD** = $(\mathbf{\Phi}, \mathbf{V}, \mathbf{E}, \{\mathbf{0}, \mathbf{1}\})$, where $\mathbf{\Phi}$ is the function node (*root*), **V** is the set of internal nodes representing the input variables, **E** is a set of edges, and $\{\mathbf{0}, \mathbf{1}\}$ are the terminal nodes. A completely specified function **F** can be specified by two sets of cubes, an *on-set* X^{on} and an *off-set* X^{off}, where $F(X^{on}) = 1$, $F(X^{off}) = 0$. Multilevel logic optimization can be seen as a procedure in which these two sets are iteratively partitioned so as to minimize some cost function (typically the number of literals). Using BDD as a platform to carry out X^{on} and X^{off} partitioning is very efficient because a set of cubes in the X^{on} (or X^{off}) of a function can be easily represented on a BDD. Function **F** is said to be *conjunctively* decomposable if its BDD can be decomposed into a quotient **Q** and a divisor **D**, such that $\mathbf{F} = \mathbf{Q} \cdot \mathbf{D}$. Function **F** is *disjunctively* decomposable if it can be decomposed into a "subtractor" **D** and a remainder **R**, such that **F = D + R**.

Definition *1* (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 **0**; and into $\mathbf{\Sigma_1}$, the set of leaf edges connected to **1**. □

Definition *2* (Paths) $\mathbf{\Pi_0}$ is the set of all paths from the *root* to *terminal node* **0**. $\mathbf{\Pi_1}$ is the set of all paths from the *root* to *terminal node* **1**. $\mathbf{\Pi} = \mathbf{\Pi_0} \bigcup \mathbf{\Pi_1}$ is the set of all paths. □

Theorem *1* (Internal Edge Property) For every internal edge $e \in (\mathbf{E} - \mathbf{\Sigma})$ there is at least one path $p \in \mathbf{\Pi_1}$ and one path $p \in \mathbf{\Pi_0}$ passing through it. □

Definition *3* (Cut) A *cut* $(\mathbf{D}, \mathbf{V} - \mathbf{D})$ of a **BDD** is a partition of its nodes **V** into disjoint subsets **D** and $(\mathbf{V} - \mathbf{D})$ such that *root* ∈ **D** and $\{\mathbf{0}, \mathbf{1}\} \subseteq$ (**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 of **D** and the support of (**V - D**) are disjoint. □

Definition *4* (Dominator) [7] Node $v \in \mathbf{V}$ which belongs to every path $p \in \mathbf{\Pi_1}$ is called a *1-dominator*. Node $v \in \mathbf{V}$ which belongs to every path $p \in \mathbf{\Pi_0}$ is called a *0-dominator*. □

It should be noted that the above definition applies only to BDDs without complement edges above v.

The presence of a *1-dominator* on a BDD indicates that the BDD can be algebraically decomposed into two conjunctive parts. Similarly, a *0-dominator* leads to an algebraic disjunctive decomposition. Although many *0-dominators* and *1-dominators* can be found in the process of decomposing a large BDD, they are rarely found on a BDD before the decomposition.

The following definition of a *generalized-dominator* is central to our AND/OR decomposition method. Under most cases, BDD decompositions are started by using this type of *dominator.*

Definition *5* (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. Let η be a set of all dangling edges. The resulting graph is called a *generalized dominator* $\mathbf{\Lambda}$. □

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.

A. Conjunctive BDD Decomposition

The following theorem shows how to obtain a Boolean divisor and perform the division by redirecting the dangling edges η of $\mathbf{\Lambda}$.

Theorem *2* (Construction of **Q**, **D**) Given a *generalized dominator* $\mathbf{\Lambda}$ of function **F**, the Boolean divisor **D** is obtained from $\mathbf{\Lambda}$ by redirecting dangling edges $e \in \eta$ of $\mathbf{\Lambda}$ to **1**. The quotient **Q** is obtained by minimizing **F** with the *off-set* of **D** as a *don't care* set. □

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 to **1**, the Boolean divisor **D** generated according to Theorem 2 covers all paths $p \in \mathbf{\Pi_1}$.

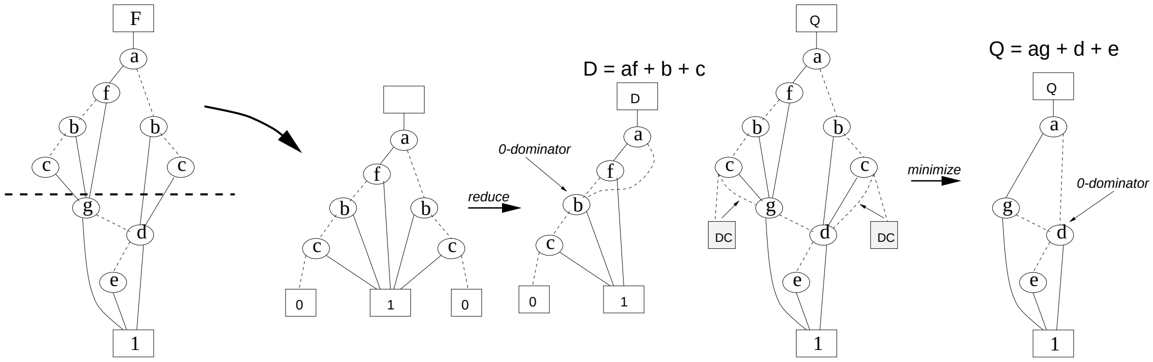
Example *1:* A complete conjunctive decomposition, including the construction of a quotient **Q**, is shown in Fig. 1. In Fig. 1(a), a *cut* is performed on the BDD. In Fig. 1(b), the *generalized-dominator* is obtained by copying the portion above that *cut* to a graph. Then a Boolean divisor is built by redirecting all the dangling edges of that graph to **1**. The reduced BDD of **D** is also shown in Fig. 1(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. 1(c), quotient **Q** is obtained from **F** by minimizing function **F** using function **D** as *don't cares.* This results in **Q** = $(ag + d + e)$. As a result of this process, the whole function can be decomposed as **F** = $(af + b + c)(ag + d + e)$. □

The following theorem is used to reduce the computation effort in the process of searching for the best decomposition.

Theorem *3* (Distinct Cuts) Reduced Boolean divisors **D** obtained from different *cuts* which contain the same subsets of $\mathbf{\Sigma_0}$ are identical. □

B. Disjunctive BDD Decompositions

Disjunctive decomposition is a dual case of the conjunctive decomposition. Here we list two most important theorems for



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

Fig. 1. Obtaining a factored form on a BDD.

this type of decomposition.

Theorem *4* (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 **0**. The remainder **R** is obtained by minimizing **F** using the *on-set* of **D** as a *don't care* set. □

Theorem *5* (Distinct Cuts) Reduced Boolean subtractors **D** obtained from different *cuts* with the same subsets of $\mathbf{\Sigma_1}$ are identical. □

III. XOR and MUX Decomposition

$\mathbf{\Sigma}$ edges defined in the previous section indicate that the value of a function can be *pre-determined* without providing the values of all variables. BDDs of functions that are mainly composed of AND/ORs tend to have many $\mathbf{\Sigma}$ edges. On the other hand, BDDs of functions populated with XORs (sometimes just one XOR) have very few or no $\mathbf{\Sigma}$ edges at all. The intuition behind this is, the values of functions with AND/ORs tend to be determined by *individual* variables. The value of functions with XORs are determined by the *relative* values of many or all variables. For example, the value of function $f = ab$ can be determined when either a or b equals to 0. But the value of function $f = a \oplus b$ will only be determined when values of both a and b are given.

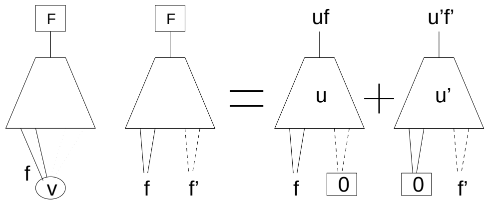
BDD decomposition based on *generalized dominators*, described in the previous section, relies on $\mathbf{\Sigma}$ edges. It is apparent that such a decomposition will fail on a BDD with few or no $\mathbf{\Sigma}$ edges. In this section, the techniques targeting XOR-type decomposition of a BDD are developed. In this case the *complement edges* are used to uncover the underlying XOR decomposition.

Complement edge was first introduced by Akers [14], and was efficiently implemented in [15]. Here we provide a brief explanation of its meaning for readers who are not familiar with BDDs. Internally, a BDD is represented by a set of *nodes*. On each *node* there are two pointers pointing to its two co-factors to maintain the connectivity. Each pointer corresponds to one edge on the BDD. *Negative pointers* take advantage of the fact that memory is allocated using the multiple of smallest unit, *byte*. Therefore, the least significant bit of any pointers is always 0. By flipping that bit to 1, the function can be seen as complementary to the function a pointer pointing to. Negative pointers correspond to *complement edges*. The canonicity of BDDs still holds if the usage of *complement edges* is restricted under certain rule. Most BDD packages restrict *complement edges* only for *else* edges. By using *complement edges*, functions which are complementary to each other are transformed into the same node on the BDD. The primary goal of introducing *complement edges* is to reduce the memory usage. Interestingly, we find that the presence of *complement edges* is related to XOR decompositions. In the sequel, we will use XNOR($\overline{\oplus}$) instead of XOR because XNOR is a more straightforward representation on BDDs.

A. Algebraic XNOR Decomposition

Definition *6* (x-dominator) Node $v \in \mathbf{V}$ which is contained in every path $p \in \mathbf{\Pi}$ is called *x-dominator*. □

Note that the definition of *x-dominator* implies that there must exist at least one *complement edge* above the *x-dominator* v. Otherwise the whole BDD section above v will collapse into v. Therefore *x-dominators* do not exist on BDDs without *complement edges*.



(a) (b) (c)

Fig. 2. Proof of Theorem 6 - algebraic XNOR decomposition

Theorem *6* (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. 2(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. 2(b). Then the BDD can be split into two disjunctive parts shown in Fig. 2(c). Note that f and f' are the *1-dominators* in their respective BDDs. By defining u to be the BDD of **F** in which v is replaced with 1, function **F** can decomposed as $\mathbf{F} = uf + u'f' = u\overline{\oplus}f$. ■

Example *2:* In Fig. 3, node x is an *x-dominator*. According to the above theorem, the BDD can be algebraically decomposed as $F = x\overline{\oplus}(u' + r' + q + y)$. □

B. 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

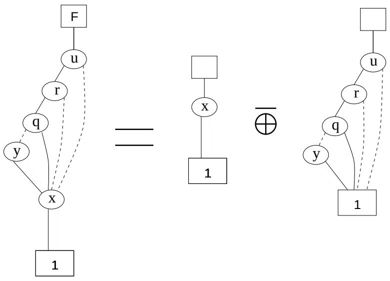


Fig. 3. Role of *x-dominator* in XNOR decomposition.

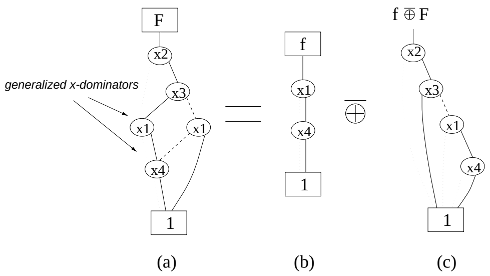


Fig. 4. XNOR decomposition of function *rnd4-1*

implementation. Usually XNOR decomposition is performed on a function in which good AND/OR decompositions are unlikely to be found.

Theorem *7* (Boolean XNOR decomposition) For a Boolean function **F**, given an arbitrary Boolean function f, there always exists a Boolean function g, such that $\mathbf{F} = f \overline{\oplus}\, g$.

Proof: The proof can be done easily using the following *lossless Boolean transformation.*

$$\mathbf{F} = f \,\overline{\oplus}\, (f \overline{\oplus} \mathbf{F}) = f \,\overline{\oplus}\, g \tag{1}$$

where f is the arbitrary Boolean function and $g = f \overline{\oplus} \mathbf{F}$. ■

We believe that there exists a set of functions f which can make the above transformation optimal in terms of the resulting logic. 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, *generalized x-dominators*.

Definition *7* (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.* □

If f is a *generalized x-dominator* on BDD **F**, by performing the transformation $g = f \overline{\oplus} \mathbf{F}$, 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 explicitly. If the transformation does not create more *XOR-related complement edges* in another part of the BDD, the transformation is called *valid*.

Example *3:* Shown in Figure 4 is the BDD of *rnd4-1* which is a test case in MCNC hard random logic suite. According to Definition 7, there are two *generalized x-dominators* on this BDD, $f = x_1 \overline{\oplus} x_4$ and $f = x_4$. We just illustrate the decomposition based on the first one. Its BDD is shown in Figure 4(b). The BDD for $g = f \overline{\oplus} \mathbf{F}$ is shown in Figure 4(c). Both f and $f \overline{\oplus} \mathbf{F}$ consist of only simple dominators. Therefore they can be further algebraically decomposed, resulting in $\mathbf{F} = (x_1 \overline{\oplus} x_4) \overline{\oplus} (x_2(x_3 + x_1 x_4))$. □

C. Functional MUX decomposition

Performing MUX decomposition with respect to a BDD variable is trivial since each node of a BDD represents a MUX natively. Usually such a straightforward MUX decomposition leads to poor multi-level Boolean expressions. However, the generalized case, *functional* MUX decomposition, where the MUX control signal is a function, instead of a single input variable, often leads to a concise expression.

Theorem *8* (Functional MUX Decomposition) Consider a structure on a BDD, in which two nodes, u and v, cover all $p \in \Pi$. The BDD can then be decomposed as $fu + f'v$, where f is obtained by redirecting node u to 1, redirecting node v to 0.

Proof: The proof is similar to that of Theorem 6. ■

Similarly 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 for various Boolean functions, they are especially common in arithmetic functions. They are frequently associated with XNOR decomposition.

Example *4:* Shown in Figure 5 is a simple example of a functional MUX decomposition. Function **F** can be decomposed as $\mathbf{F} = fc + f'd$, where $f = a' + b$ serves as a MUX control signal.

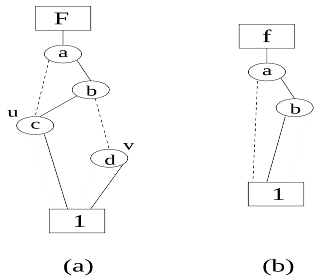


Fig. 5. Example of functional MUX decomposition: $\mathbf{F} = fc + f'd$

IV. Implementation and Results

A new logic optimization tool, *BDDlopt*, which implements the techniques presented in this paper, has been developed in C and tested on a large number of benchmark circuits. CUDD package is used to carry out BDD operations.

Basically, a BDD is iteratively decomposed by finding the best decomposition at each iteration step. The structural information of a BDD is obtained by a single fast scan on the BDD. All simple dominators (*0, 1* and *x*) and *functional MUX decomposition* are found during the scan. *XOR-related complement edge* information is also obtained. If a simple dominator or a functional MUX decomposition is found, the decomposition is taken immediately. Otherwise, the BDD will be decomposed by either a *generalized-dominator* or a *generalized x-dominator*. The choice between these two depends on the information collected by the BDD scan.

The cost function used for the *generalized-dominator*-based decomposition can be calculated as follows:

$$Cost = \alpha \frac{|V| + |D|}{|V_F|} + (1-\alpha)\frac{S_I}{S_F} \tag{2}$$

Here $|V|$ is number of nodes in **Q** (or **R**); $|D|$ is the number of nodes of the Boolean divisor (or subtractor) **D**; $|V_F|$ is the number of nodes in the original BDD of **F**; S_I is the size of the intersection of the support of **Q** and **D** (or **D** and **R**); S_F is the size of the support of **F**; and α is a predefined weight. In this experiment α is set to 0.5.

Additional XOR-related *complement edge* information is considered in the cost function used by *generalized x-dominator*-based decomposition. Usually, due to the correlation between the number of nodes and the number of variables on a BDD, if a function has a good decomposition, the cost of that decomposition will be significantly lower than that of others.

A set of factoring trees is built along with the BDD decompositions. After decompositions of all the output functions have been finished, BDDs are rebuilt on those factoring trees to find all possible logic sharing. No other attempt is used to extract the sharing between primary outputs. We believe that identifying sharing between output functions before doing decomposition will further improve the quality of synthesized results. CPU time is also going to improve, because the shared BDDs are decomposed only once.

The experiments have been conducted on UltraSPARC-5/320M. They cover most of the IWLS'91 and part of the IWLS'93 combinational test cases. All the test cases are 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* has been 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 SIS mapper and *ceres* [16] 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.

For AND/OR intensive circuits, shown in Table I, *BDDlopt* uses slightly fewer gates than SIS, and more area than SIS in *Total*. The slight increase in area is due to the higher cost of XOR gates. On average, the final synthesis results using *BDDlopt* and SIS on this class of functions are almost the same. Near optimal results are obtained from both SIS and *BDDlopt*, although *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. *ceres* generates better mapping results that SIS mapper. Unfortunately, *ceres* was not stable on several circuits which makes the complete comparison difficult. Only results of SIS mapper are presented. The results of a techniques targeting specifically at XOR decomposition [3] 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.* [3]. 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.

V. Discussion

In this section we discuss (in an attempt to understand) two issues which are crucial to the success of the proposed decomposition method.

A. Role of variable reordering

Most variable reordering algorithms are based on swapping of the adjacent variables while preserve the functionality on all nodes. Let us look at a snapshot of a variable swapping [17]. Shown in Fig 6(a) is a function, $F = (x_i, (x_{i+1}, F_{11}, F_{10}), (x_{i+1}, F_{01}, F_{00}))$. By swapping x_i and x_{i+1}, the function can be equivalently represented as $F = (x_{i+1}, (x_i, F_{11}, F_{01}), (x_i, F_{10}, F_{00}))$. This variable swapping creates new opportunity to minimize the BDD. Let us consider

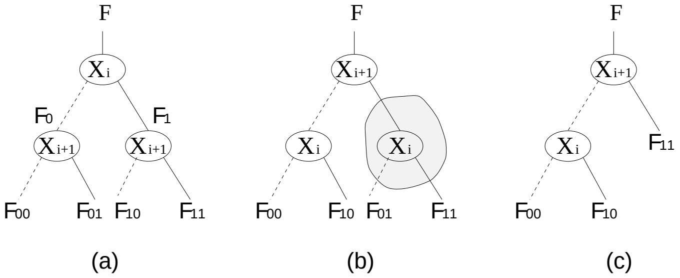


Fig. 6. A snapshot of variable swapping

the case when $F_{01} = F_{11}$. As a result of variable swapping, a *degenerate node* (shadowed part of Fig 6(b)) can be removed, resulting in the reduced BDD in Fig 6(c). The Boolean representations based on the original BDD in Fig 6(a) is $F = x_i(x_{i+1}F_{11} + x'_{i+1}F_{10}) + x'_i(x_{i+1}F_{01} + x'_{i+1}F_{00})$. Considering that $F_{01} = F_{11}$, the expression for F can be simplified to $F = X_{i+1}F_{11} + x'_{i+1}(x_iF_{10} + x_iF_{00})$ by using classical factorization technique. However, the same form can be obtained directly from the BDD in Fig 6(c). Often, the variable reordering algorithms can provide a globally minimized BDD. Therefore, BDD variable reordering can be seen as an implicit logic minimization process. A good Boolean representation can be obtained by carefully examining the structure of the reordered BDD. We observe that, if a function can be decomposed algebraically, the reordering algorithms (specifically *sifting*) can always reveal that decomposition in the form of simple dominators (*0, 1* and *x-dominator*).

In this sense, the final Boolean representation is affected by the number of nodes in a BDD, and different variable reordering algorithms will generate different results. Shown in Table III are the statistical results of *BDDlopt* for 56 combinational circuits selected from *MCNC* benchmarks. The correlation between the number of BDD nodes and the quality of final synthesis results is obvious. The performance of *Sifting*-based algorithms is far better than *window permutation*-based algorithms. But, interestingly, while the number of BDD nodes is strongly affected by reordering algorithms (*normalized standard deviation* σ = 162%), the final synthesis results are much less sensitive to different algorithms (*normalized* $\sigma = 16\%$). Probably the final sharing extraction procedure efficiently removes many duplications. Overall, *Sifting* outperforms other algorithms in terms of the synthesis quality and the CPU time. Note that usually *sifting* generates smaller BDDs than *window permutation*-based algorithms and uses more time. However when those algorithms are plugged into the application like *BDDlopt*, *sifting* is also the most efficient algorithm in terms of the CPU time. This is probably because fewer BDD nodes means fewer decompositions.

B. BDD Minimization

BDDlopt relies on BDD minimization algorithms to carry out Boolean division and subtraction described in this paper. Once a Boolean divisor (subtractor) **D** is generated from a *generalized dominator*, quotient **Q** (remainder **R**) can be obtained by *restrict*-based BDD minimization functions. $\mathbf{Q} = \mathbf{F}/\mathbf{D}$; $\mathbf{R} = \mathbf{F}/\mathbf{D}'$.

Minimization of BDDs with *don't cares* has been a research topic for several years [18], [19], [20], [21]. It was mainly used for state space traversal in sequential verification. Therefore the efficiency of these algorithms only affects computation time, not the final results. In logic synthesis, the BDD size is crucial to the

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 [3]
Name	In	Out	Lit.	gates	area	CPU	Lit.	gates	area	CPU	XORs	gates
5xp1	7	10	132	81	195	4.1	95	67	172	0.4	4/16	66
9sym	9	1	274	152	396	22.0	70	42	109	1.0	0/4	64
9symml	9	1	186	102	270	19.7	70	41	108	0.9	0/4	-
alu2	10	6	361	217	524	74.7	318	230	632	2.8	13/53	-
alu4	14	8	694	409	996	286.3	930	582	1655	15.9	23/124	-
cordic	23	2	64	34	94	0.9	56	47	126	0.5	6/16	-
f51m	8	8	98	58	139	9.0	73	56	174	0.3	5/11	63
my_add	33	17	192	156	287	3.1	128	110	286	8.9	16/32	113
parity	16	1	60	15	75	0.6	16	15	75	0.1	15/15	15
rd53	5	3	34	22	47	1.3	38	25	72	0.2	3/6	25
rd73	7	3	189	106	258	12.1	80	45	133	0.8	5/8	41
rd84	8	4	348	192	468	42.8	115	62	189	1.4	6/12	66
t481	16	1	881	407	1023	208.6	16	15	45	0.3	5/5	23
z4ml	7	4	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 [3]. Since *ceres* is not stable on this class of functions, SIS mapper is used. Circuits are mapped to *mcnc.genlib*. The number of XORs after/before technology mapping is shown in *XORs* column.

Reorder Algorithm	Literals	Nodes	CPU(s)
SIFT	8808	23925	199.9
SIFT_CONVERGE	9065	24438	209.9
GROUP_SIFT	8936	23516	206.6
GROUP_SIFT_CONV	8842	23024	221.0
SYMM_SIFT	8897	23850	206.0
SYMM_SIFT_CONV	8893	23447	206.4
WINDOW2	11849	941622	672.9
WINDOW2_CONV	14255	339351	1668.2
WINDOW3	11360	297340	325.0
WINDOW3_CONV	11277	66213	282.9
WINDOW4	10689	147788	278.1
WINDOW4_CONV	10637	51672	265.4
Normlized σ	16%	162%	

TABLE III

BDDlopt-1.2.5 RUN RESULTS WITH DIFFERENT VARIABLE REORDERING ALGORITHMS. *Literals* IS THE TOTAL NUMBER OF LITERALS OF 56 CIRCUITS AFTER SYNTHESIS. *Nodes* IS THE SUM OF ALL INITIAL AND INTERMEDIATE BDD NODES. *CPU* IS THE TIME TO RUN ALL 56 TEST CASES.

quality of final results. These algorithms must be re-evaluated extensively. The same concern was raised elsewhere [22].

VI. Future Work

BDDlopt was created to verify our BDD decomposition-based logic optimization methodology. It needs to be further polished to handle large test cases. Currently, only monolithic (global) BDDs are used for decomposition. Therefore the method fails to generate good results for large circuits, whose BDD size is not manageable. This is especially true for some arithmetic circuits; while the size of global BDD is several orders of magnitude larger than that of local BDD obtained by partitioning the initial network. Application of the proposed method to partitioned networks is currently under active research.

A foreseeable application of the proposed method is synthesis of mixed CMOS-PTL circuits. CMOS and PTL are complementary to each other in terms of the efficiency in implementing the AND/OR and XOR or MUX logic. Efficient combination of these two will result in smaller, faster and more energy-efficient circuit implementations. Since the proposed method can efficiently handle both AND/OR, and XOR/MUX decompositions, it opens the possibility for CMOS-PTL mixed design style. Preliminary results [23] show that the average cost of CMOS/PTL circuits for XOR-intensive test cases is only 57% of static CMOS-only circuits generated by SIS (*script.rugged*).

References

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

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

[4] Y. Ye and K. Roy, "A Graph-Based Synthesis Algorithm for AND/XOR Networks," in *Proc. Design Automation Conference*, 1997, pp. 107–112.

[5] S. Chattopadhyay, S. Roy, and P. Chaudhuri, "KGPMIN: An Efficient Multilevel Multioutput AND-OR-XOR Minimizer," *IEEE Trans. on CAD*, vol. 16, no. 3, pp. 257–265, March 1997.

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

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

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

[9] 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.

[10] U. Kebschull, E. Schubert, and W Rosenstiel, "Multilevel logic synthesis based on functional decision diagrams," in *European Conference on Design Automation*, 1992, pp. 43–47.

[11] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski, "Efficient representation and manipulation of switching functions based on order kronecker function decision diagrams," in *Proc. Design Automation Conference*, 1994, pp. 415–419.

[12] F. Somenzi, *CUDD: CU Decision Diagram Package*, `ftp://vlsi.colorado.edu/pub/`.

[13] C. Yang and M. Ciesielski, "Logic Optimization on BDDs," Tech. Rep. TR-CSE-98-05, Department of Electrical and Computer Engineering, University of Massachusetts Amherst, 1998.

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

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

[16] 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.

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

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

[19] 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.

[20] 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.

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

[22] 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.

[23] C. Yang and M. Ciesielski, "Synthesis for Mixed CMOS/PTL Logic: Preliminary Results," in *Submission to Intl. Workshop on Logic Synthesis*, 1999.